

# THE EFFECTIVE APPLICATION OF SYNTACTIC MACROS TO LANGUAGE EXTENSIBILITY

William R. Campbell

A Thesis Submitted for the Degree of PhD  
at the  
University of St Andrews



1978

Full metadata for this item is available in  
St Andrews Research Repository  
at:

<http://research-repository.st-andrews.ac.uk/>

Please use this identifier to cite or link to this item:

<http://hdl.handle.net/10023/13411>

This item is protected by original copyright

# The Effective Application of Syntactic

## Macros to Language Extensibility

### Abstract

Starting from B M Leavenworth's proposal for syntactic macros, we describe an extension language  $L_E$  with which one may extend a base language  $L_B$  for defining a new programming language  $L_P$ . The syntactic macro processor is designed to minimise the overheads required for implementing the extensions and for carrying the syntax and data type error diagnostics of  $L_B$  through to the extended language  $L_P$ . Wherever possible, programming errors are flagged where they are introduced in the source text, whether in a macro definition or in a macro call.  $L_E$  provides a notation, similar to popular extended forms of BNF, for specifying alternative syntaxes for new linguistic forms in the macro template, a separate assertion clause for imposing context sensitive restrictions on macro calls which cannot be imposed by the template, and a non-procedural language which reflects the nested structure of the template for prescribing conditional text replacement in the macro body. A super user may use  $L_E$  for introducing new linguistic forms to  $L_B$  and for redefining, replacing or deleting existing forms. The end user is given the syntactic macro in terms of an  $L_P$  macro declaration with which he may define new forms which are local to the lexical environments in which they are declared in his  $L_P$  program. Because the macro process is embedded in and directed by a deterministic top down parse, the user can be sure that his extensions are unambiguous.

Examples of macro definitions are given using a base language  $L_B$  which has been designed to be rich enough in syntax and data types for illustrating the problems encountered in extending high level languages. An implementation of a compiler/processor for  $L_B$  and  $L_E$  is also described. A survey of previous work in this area, summaries of  $L_E$  and  $L_B$ , and a description of the abstract target machine are contained in appendices.

ProQuest Number: 10167249

All rights reserved

INFORMATION TO ALL USERS

The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if material had to be removed, a note will indicate the deletion.



ProQuest 10167249

Published by ProQuest LLC (2017). Copyright of the Dissertation is held by the Author.

All rights reserved.

This work is protected against unauthorized copying under Title 17, United States Code  
Microform Edition © ProQuest LLC.

ProQuest LLC.  
789 East Eisenhower Parkway  
P.O. Box 1346  
Ann Arbor, MI 48106 – 1346

THE EFFECTIVE APPLICATION OF SYNTACTIC  
MACROS TO LANGUAGE EXTENSIBILITY

by

William R Campbell

A thesis submitted for the degree of Doctor of Philosophy

Department of Computational Science  
University of St. Andrews  
St. Andrews, Scotland

April 1978





Th 9101

### Preface

I hereby declare that this thesis has been composed by myself; that the work of which it is a record has been done by myself; and, that it has not been accepted in any previous application for any higher degree. This research concerning the effective application of the syntactic macro was undertaken on 1st October 1974, the date of my admission as a research student under Ordinance General No. 12 for the degree of Doctor of Philosophy (Ph D).

William R Campbell

(iii)

I hereby declare that the conditions of the Ordinance and Regulations for the degree of Doctor of Philosophy (Ph D) at the University of St. Andrews have been fulfilled by the candidate, William R Campbell.

Professor A J Cole

### Acknowledgements

I am indebted to my research supervisor and friend, Professor A J Cole, whose kind criticisms and suggestions guided the development of the syntactic macro processor. I should like to thank Mr Tony Davie who will always ask the most awkward questions at the correct time. I should also like to acknowledge the invaluable criticisms made by Dr Phillipe Jorrand during his visit to St Andrews in October of 1976. Finally, I am indebted to Mrs Margaret Buchanan for her careful typing of the final draft.

## CONTENT

	Page
Preface .....	ii
Acknowledgements .....	iv
Content .....	v
 Chapter	
1. Introduction .....	1
2. The Syntactic Macro .....	10
2.1 Preliminary notions .....	10
2.1.1 The underlying grammar .....	10
2.1.2 The base language $L_B$ .....	12
2.2 The macro .....	12
2.2.1 Type 1 macros - a restricted form .....	13
2.2.2 Type 2 macros - conditional text replacement	16
2.2.2.1 Type 2 macro templates .....	16
2.2.2.2 Type 2 macro bodies .....	19
2.2.2.3 A revision to the macro process ...	24
2.3 Deleting and replacing $L_P$ forms .....	26
2.3.1 $L_P$ deletions .....	26
2.3.2 Replacing $L_P$ forms .....	27
2.4 Interpretation of the macro process .....	29
3. Semantic Actions in the Macro Process .....	31
3.1 Computation trees .....	31
3.2 Diagnosing type violations in actual parameters ...	37
3.2.1 Type restrictions specified in the template	39
3.2.2 The <u>where</u> clause .....	42
3.2.3 Effects on the user's interpretation .....	49

3.3	Diagnosing type violations in a macro body .....	52
3.3.1	Type violations in a type 1 macro body .....	52
3.3.2	Type violations in a type 2 macro body .....	55
3.3.3	Nested macro calls .....	57
3.3.4	Computing addresses in the macro body .....	59
3.4	Generic macros .....	61
3.4.1	Expansion conditional upon predicates .....	61
3.4.2	Generic declarations in $L_B$ .....	63
3.5	The complete macro process .....	65
4.	Scope of Definition .....	70
4.1	The influence of lexical scope.....	70
4.2	The definition of $L_P$ .....	72
4.3	Macro definitions in $L_P$ .....	73
5.	Implementation .....	77
5.1	Introduction .....	77
5.2	Representing the underlying syntax .....	78
5.2.1	Lexical analysis - the micro syntax .....	78
5.2.2	Parsing $L_E$ .....	79
5.2.3	Parsing $L_B$ .....	80
5.3	The compiler for $L_B$ .....	82
5.4	The macro processor .....	88
5.4.1	Interpreting the macro definition .....	88
5.4.1.1	The macro template .....	88
5.4.1.2	The assertion .....	90
5.4.1.3	The macro body .....	90
5.4.1.4	Modifying the underlying grammar ..	95
5.4.2	Interpreting macro calls .....	96
5.4.2.1	Recognition .....	96
5.4.2.2	Assertion verification .....	99
5.4.2.3	Macro expansion .....	100

5.5	Alternative modifications to $L_P$ .....	102
5.5.1	Replacements .....	102
5.5.2	Deletions .....	104
5.5.3	Macro declarations .....	105
5.6	Generic declarations .....	108
5.7	Summary .....	109
6.	Topics Related to the Syntactic Macro .....	110
6.1	Recursive macro definitions .....	110
6.2	Nested macro definitions .....	111
6.3	Generalising $L_B$ .....	115
6.4	Syntactic macros in an LR parse .....	117
7.	Conclusion .....	119
7.1	Contributions .....	119
7.2	Possible modifications and areas for further investigation .....	122
7.3	Syntactic extensibility .....	124
Appendix		
1	Previous Work Concerning the Syntactic Macro .....	127
A1.1	McIlroy's macros .....	127
A1.2	The syntactic macro facility .....	128
A1.2.1	Cheatham's macros .....	128
A1.2.2	The Schuman and Jorrand proposal .....	130
A1.2.3	Hammer's alternative to the string replacement process .....	132
A1.3	Applications to syntactic extensibility .....	133
A1.3.1	Definitions in Algol .....	134
A1.3.2	ELF - an extensible language facility ....	135
A1.3.3	Vidart's extensions to GSL .....	136
A1.3.4	ALEC - a language with an extensible compiler .....	137

A1.3.5	Syntactic extensions in BALM .....	140
A1.3.6	ECT - an extensible contractible translator .....	142
A1.4	Summary .....	144
2	Summaries of $L_B$ and $L_E$ .....	147
A2.1	The base language $L_B$ .....	147
A2.1.1	Syntax .....	147
A2.1.2	Explanations of various constructions ....	150
A2.2	The extension language $L_E$ .....	152
A2.2.1	An $L_E$ program .....	152
A2.2.2	A macro definition .....	152
A2.2.2.1	The macro template .....	153
A2.2.2.2	The assertion .....	154
A2.2.2.3	The macro body .....	156
A2.2.2.3	Predicates and formal parameter references .....	158
A2.2.3	Replacements and deletions .....	159
A2.3	Lexical representations - the micro syntax .....	159
A2.4	Two additional examples .....	160
3.	The Object Machine .....	162
References	.....	167
Example Listings	.....	170



## CHAPTER 1

### Introduction

Syntactic macros were first introduced for extending high level languages in two separate papers published by Cheatham (1966) and Leavenworth (1966). In a later paper Schuman and Jorrand (1970) propose a more ambitious version of the syntactic macro mechanism. Since its introduction the syntactic macro has been applied as a basis for many extensible languages. Some of these mechanisms and their applications are discussed in a survey in Appendix 1 to this thesis.

The syntactic macro permits what Standish (1975) has called 'paraphrase' extensibility since new linguistic forms are defined in terms of old forms. This sort of mechanism lends itself to introducing new syntactic constructs into a language rather than to the definition of new data types permitted in a language like Algol 68 (van Wijngaarden, 1975). We shall confine ourselves here to the syntactic macro; a survey of language extension mechanisms of all types may be found in a paper by Solntseff and Yezerski (1974).

The syntactic macro differs from the conventional text macro in that it is embedded in and directed by a compiler's syntax analyser. Its constituents, eg its template, parameters and body, are syntactic entities in the language being extended.

We start with a base language  $L_B$  consisting of a set of basic programming constructs, say a subset of Algol, plus a syntactic macro facility.  $L_B$ 's context free syntax may be described by a sequence of BNF production rules satisfying the restrictions imposed by a particular class of grammars, eg one of the LL or LR classes. We may then define new BNF rules using syntactic macros and so extend  $L_B$  into a new programming language  $L_P$ . The newly defined rules in  $L_P$  are invoked by macro calls. As these new rules must satisfy the same grammatical restrictions imposed on  $L_B$ , the analyser can recognise calls in the context of the extended grammar. Subsequent macro definitions may employ nested macro calls for

effecting further extensions so  $L_p$ , initially equivalent to  $L_B$ , may be continually extended.

Our aim is to illustrate how the original proposals may be modified for making the syntactic macro more effective for extending high level languages. Briefly, our objectives are

1. to increase the macro's flexibility with a widened syntactic domain of definable linguistic forms and a means for replacing and deleting forms;
2. a more structured macro template for prescribing alternative syntaxes for macro calls and a non-procedural macro time language, reflecting the template's structure, for expressing conditional text replacement;
3. more effective context free and context sensitive error diagnostics - whose corruption is common to most macro based extension mechanisms; and
4. a means by which the user may control the scope of his syntactic macro definitions.

In order to clarify these objectives we shall first discuss the original Leavenworth proposal. A discussion of other proposals, including those of Cheatham, Schuman and Jorrand, may be found in Appendix 1.

Leavenworth proposes two kinds of macros, smacros and fmacros for defining new statement forms and new primary function forms respectively. The macro process itself is embedded in a top down deterministic grammar. An smacro has the form,

(1.1) smacro <macro template> define <macro body> endmacro

The macro template prescribes the syntax of calls to the macro and may optionally contain formal parameters. Each macro template, and so calls to the macro, must start with a unique basic token. The macro body, written in the current  $L_p$ , prescribes the replacement text which is to be substituted for a macro call. If the template contains formal parameters then the actual parameters supplied in a macro call are copied into the



parameters substituted for corresponding parameter markers. For example, (1.4) would be replaced by

```
(1.5)  begin
        i:=1;
        L1: if i <= n+1 then
              begin
                a(i) :=0; i:=i+1;
              goto L1
              end
        end
```

Any remaining nested calls (we assume none in (1.5)) are similarly expanded before the resultant  $L_B$  text is analysed and compiled to some abstract or machine code. Of course, the  $L_B$  text finally produced by an smacro call must satisfy the syntax for a statement since it will be analysed as such.

Though macro calls are recognised syntactically, their effect is the replacement of one string by another. The underlying compiler itself deals strictly with  $L_B$  text.

Leavenworth's templates may also contain both optional sequences (of tokens and parameters) and lists of alternative sequences, one of which must be matched in a call. The manner in which such sequences are matched (or not matched) directs conditional text replacement at call time. For example, consider another smacro from Leavenworth.

```
smacro
  for var := express {while express1 [by express2] to express3}
  do statement
define
  begin L1 : $1 := $2;
(1.6)  L2 : if {1 $3 then begin $6; goto L1}
        {3 $1 <= $5 then begin $6;
          $1 := $1 + {2 $4} {¬2 1}; goto L2}
        end
  end
endmacro
```

In the template, square brackets (`[,]`) enclose optional sequences and braces (`{,}`) enclose a list of alternative sequences separated by the vertical bar (`|`). Optional sequences and alternative sequences are called groups; each group is numbered from left to right. To each group in the template corresponds a group in the macro body, enclosed with braces and numbered with a corresponding ordinal. If a group in the template is matched at call time then the corresponding group in the body is copied for replacement. The notation `{¬n...}` means that the body group is copied if and only if the  $n^{\text{th}}$  group in the template is not matched. For example, a call to (1.6) such as

(1.7) for i := 1 by 3 to n do X := X+a(i)

would expand to the following  $L_B$  text.

```

begin L1: i:=1;
          L2: if i <= n then begin X := X+a(i);
(1.8)                                i:=i+3; goto L2
                                     end
end

```

Just as with calls to simpler smacros, all possible calls to a smacro with groups must produce  $L_B$  text which satisfies the syntax of statement in order to permit correct analysis. fmacros, which introduce new primary function forms having values, are processed like smacros. Calls to fmacros may appear anywhere primary values are allowed and the  $L_B$  text produced by such calls must have the syntax of primary values.

Since its introduction the syntactic macro has found favour as a basis for language extension for two reasons. Firstly, its syntactic nature clarifies that class of syntactic rules which may be added to an underlying grammar. Secondly, its string replacement nature allows a simpler expression of the meanings of new constructs in the language being extended; one need not learn yet another systems programming language for making definitions. But an examination of Leavenworth's original proposal does suggest areas for improvement.

The first area for improvement comes under the heading of generality. We needn't restrict the macro to the definition of just statements or primary functions. We might just as well allow definitions for any syntactic entity in  $L_B$ , eg declarations. Nor need we insist that macro templates start with unique tokens if we restrict our grammar to some deterministic (LL(1) or LR(1)) class. So long as the new BNF rules preserve the chosen deterministic class, the syntax analyser can recognise macro calls in the extended grammar. Finally, 'extensibility' should not restrict us to macros for introducing new linguistic forms but should encompass replacement and even deletion of existing forms.

A second area for improvement concerns the specification of replacement text conditional upon alternative syntaxes scanned in the macro template. Leavenworth's notation for defining groups has little structure and allows contradiction in the prescription of text replacement. For example, assuming the groups in the template for smacro (1.6), one is not prevented from defining the following corresponding group in the body.

(1.9) {¬2 ...\$4 ...}

Expression (1.9) says that the text, "...\$4..." is copied for replacement if the second template group, containing the parameter to which \$4 refers, is not scanned in a macro call. We might ask for more structure in the notation for groups in the template and for a macro time language which reflects this structure for prescribing conditional text replacement in the body. The structures chosen should facilitate a more complex nesting of alternative syntaxes in the macro template and must prohibit contradictory expressions like (1.9) in the macro body. Finally, the notation for groups might be extended to handle the repeated scanning of lists supplied in macro calls.

The way in which Leavenworth's syntactic macros are processed suggests a third area for improvement. String substitution dictates

that the underlying compiler deals only with  $L_B$  text after all macro calls have been evaluated. Cole (1976) makes two points here:

1. The process demands much repetitive scanning of source text.  
A macro's body is scanned as many times as the macro is called.  
Each actual parameter to a call is scanned as many times as it is substituted for a formal parameter marker in a macro body.
2. Delaying syntax analysis until all macro calls have been evaluated prohibits meaningful syntax error detection since error messages refer to the  $L_B$  text produced by evaluation rather than to the text of the extended  $L_P$ . It is not always evident where the errors were introduced, whether in the defining text of a macro body or in an actual parameter to a macro call.

The inefficiency introduced by the repeated scanning of source text is not merely a micro-inefficiency resulting from a coding decision but, excusing the pun, a macro-inefficiency inherent to the processing algorithm. Minimising such repetitive scanning would be worth our while. The perversion of syntax error diagnostics in the extended  $L_P$  is a greater disadvantage from the user's point of view. If our base language  $L_B$  has type matching rules then the detection of mismatched types is similarly corrupted in the extended  $L_P$  by the string substitution process. Surely we should require that any context free (syntax) or context sensitive (type) error diagnostic facilities available to  $L_B$  be carried through to the extended  $L_P$ . Hammer (1971) and Cole (1976) suggest that we partially compile both macro bodies at macro definition time and actual parameters at macro call time. In our search for a more effective syntactic macro we shall be discussing where such partial compilation may be applied to the macro process, how this application must affect the user's string substitution interpretation of the macro process and where early compilation can both minimise the repeated scanning of source text and maximise effective error diagnosis.



Finally, a fourth area for improvement concerns the textual scoping of macro definitions. We intend to distinguish between two kinds of macro definition and the nature of extensibility afforded by each. In extending  $L_B$  to a new language  $L_P$  all macro definitions precede any  $L_P$  program and no macro body may contain free variables declared in the  $L_P$  program. We can also give the syntactic macro to the end user, ie to the  $L_P$  programmer, as a special form of declaration with which he can modify  $L_P$  throughout his program. The scope of such a macro would be the block in which it is declared and the macro's body might contain variable names declared globally to it. Since this macro would be an  $L_B$  construct, the user defining  $L_P$  has the choice of retaining the macro definition facility in  $L_P$  or deleting it so as to fix  $L_P$  permanently.

Many of our objectives are inspired by the work of others. Rather than referring to that work here we shall refer to it in subsequent chapters as we propose our own solutions and survey it as a whole in Appendix 1.

As indicated above, our primary interests are not in increasing the syntactic macro's definitional power; in fact we are less ambitious than Schuman and Jorrand (1970) in this respect. Our overall objective is to increase the macro's effectiveness as a practical tool for extending high level languages. In meeting the objectives outlined in this introduction we aim to make the syntactic macro and, more generally, extensibility more palatable to the average high level language programmer.

We propose a syntactic macro facility for meeting these objectives in Chapters 2, 3 and 4. We begin in Chapter 2 by introducing an underlying grammar, some lexical notions and an Algol like base language  $L_B$  which we shall use for illustrating our facility. We then introduce our syntactic macro and a non-procedural language  $L_E$  for describing macro definitions. We also outline a processing algorithm permitting more



effective (context free) syntax error diagnosis and introduce  $L_E$  constructs for deleting and replacing  $L_P$  forms. We treat (context sensitive) type checking separately in Chapter 3 and introduce further  $L_E$  constructs for prescribing type diagnostics. We discuss some implications of type diagnosis to the user's interpretation of his definitions and then describe a processing algorithm which takes types into account. In Chapter 4 we discuss the scope of definitions and distinguish between definitions for extending  $L_B$  to a new language  $L_P$  and definitions which the end user may write in his own  $L_P$  programs.

We discuss some of the more salient features of an implementation of our extension facility in Chapter 5. In Chapter 6 we discuss some features which might have been included in our facility but which we have chosen to exclude. We draw discussion of our facility, as we have defined it, to a close in Chapter 7.

Work done by others in the field of syntactic macros is surveyed in Appendix 1. Appendix 2 is a summary of an illustrative base language  $L_B$ , and the extension language  $L_E$ . Appendix 3 contains a brief description of an abstract target machine for which we compile  $L_P$  programs.

## CHAPTER. 2

### The Syntactic Macro

#### 2.1 Preliminary notions

Before introducing our syntactic macro we first characterise the underlying grammar in which we shall embed our macro and introduce the base language which we shall use for illustrating macro definitions.

##### 2.1.1 The underlying grammar

The syntactic macro may be embedded in any type of context free grammar so long as  $L_B$  rules and macro calls can be easily recognised. Leavenworth sets his macros in a top down parse (driven by prefixing tokens) and Cheatham sets his in a (bottom up) simple precedence analysis of the type described by Wirth and Weber (1966). We set our macro facility in a deterministic top down grammar similar to the LL(1) grammars described by Knuth (1971).

In a top down parse one is attempting to satisfy a target syntactic entity, initially <program>, in the context of a left to right scan of the source text. A target syntactic class (nonterminal) reduces via a BNF rule to a compound target consisting of an ordered sequence of sub-targets which in turn must be satisfied. A target basic symbol (terminal) is satisfied by scanning that symbol in the source text. For example, if our grammar were to contain the BNF rule

(2.1) <statement> ::= if <expression> then <statement>

then the target (or sub-target) <statement> might reduce to the compound target

(2.2) if, <expression>, then, <statement>.

We would satisfy (2.2) by scanning over the 'if' in the source text, satisfying <expression>, scanning over the 'then' and finally satisfying <statement>. If we cannot scan a basic symbol in the source text then we have a syntax error.

A target syntactic class may reduce to more than one compound target if several BNF rules are defined for the syntactic class. Simply, the LL(1) condition states that the analyser must always be able to decide which BNF rule to follow in making a reduction on the basis of the first unscanned symbol in the source text. This decision is trivial if all BNF rules start with basic symbols but is more complicated where rules start with syntactic classes or produce the empty string. A more complete discussion of the LL(1) decision process may be found in the Knuth paper.

We alter the LL(1) scheme to take account of the specification of alternative syntaxes in the macro template; this alteration is explained later in this chapter where subtemplates are introduced. For now it is sufficient to say the domain of definable forms in our scheme is similar to that for the LL(1) scheme.

We have chosen a deterministic top down parsing scheme since its deterministic quality eliminates ambiguity, alternative syntaxes are more clearly specified in the macro template and, in our view, top down analysis conforms more closely to the notion of macro expansion than does bottom up analysis. Further, the 'naive' user can safely avoid LL(1) conflicts by prefixing his templates with unique identifying basic symbols while the more sophisticated user may make use of the broader interpretation of the LL(1) condition in defining his templates. Of course, any implementation of our processor must flag templates which do not conform to the condition.

Both the syntactic macro processor and the underlying compiler deal with tokens constructed from input text. The tokenisation of input to our system is compatible with the tokenisation of input in the sort of high level languages we seek to extend. Tokens are either simple basic symbols, eg reserved words, or special tokens, eg constants and identifiers, with semantic values in either the base language  $L_B$ , the macro time extension language  $L_E$  or both. In our scheme the

alphabet of simple basic symbols may be modified by the introduction of new symbols in macro templates but the alphabet of special tokens is permanently fixed. The micro syntax which describes the forms of both types of tokens is given in Appendix 2.

### 2.1.2 The base language $L_B$

For illustrating our macro facility we have defined a simple Algol like base language  $L_B$  whose syntax is described by a deterministic top down grammar.  $L_B$  is defined independently of the macro time extension language and includes such features as type declarations, nested scope, block expressions and a few simple control constructs. Any constructs easily expressed in terms of other  $L_B$  constructs have been omitted since they may be defined as extensions. Some popular high level constructs, eg procedures, have been left out to keep  $L_B$  to a manageable size. At the same time  $L_B$  is rich enough in syntax and data types, including structures and pointers, for illustrating the various characteristics of the macro extension facility. The meanings of most  $L_B$  constructs should be evident from their use in the macro definition examples; a description of  $L_B$  as a whole, including its context free grammar, may be found in Appendix 2.

## 2.2 The macro

The syntactic macros we introduce here may be grouped into two types determined by the definitional flexibility each allows and by how their definitions and calls are processed. Though the user need not differentiate between these types initially, the processor does so we shall describe them in this context. In this chapter we discuss only the context free syntactic extension capabilities of our facility; we shall discuss the macro's more context sensitive type checking capabilities separately in Chapter 3. Those linguistic constructs we use for describing macro definitions form the macro time extension language  $L_E$ . We describe the various  $L_E$  constructs as they are required

in the definitions;  $L_E$  is briefly described as a whole in Appendix 2.

### 2.2.1 Type 1 macros - a restricted form

A type 1 macro definition has the general form,

(2.3) define <target class> rule <macro template>  
          means <macro body> endef

where the target class names that syntactic class for which a new alternative form, specified by the macro template, is to be defined. The remaining constituents of the type 1 macro definition obey the following restrictions:

1. The macro template is a sequence of quoted basic symbols and formal parameters denoted by (possibly subscripted) syntactic class names. Actual parameters supplied for the formal parameters at call time must satisfy the syntaxes specified in the syntactic class names.
2. The macro body is a bracketed string of  $L_P$  source text, where  $L_P$  is  $L_B$  plus any previous extensions, which may contain references to the formal parameters in the template. Taking the syntactic classes of the formal parameters into account, the  $L_P$  text in the macro body must satisfy the syntax of the target class.
3. No conditional text replacement may be prescribed.

For example, consider the following type 1 macro definition,

```

define $statement
  rule 'while' $expression1 'do' $statement1
means
  [ begin
    label l1;
    (2.4) lab l1: if $expression1 then
              begin
                $statement1;
                goto.l1
              end
    end ]
endef
```

Each syntactic class name, denoting either a target class (eg \$statement) or a formal parameter (eg \$expression<sub>1</sub> and \$statement<sub>1</sub>), is appended on the left with a '\$' symbol. Formal parameters may be subscripted to differentiate among different parameters of the same syntactic class. A macro template need not start with a quoted basic symbol so long as the deterministic quality of L<sub>P</sub>'s top down grammar is preserved.

The syntactic macro definition (2.4) effectively introduces a new BNF rule,

(2.5) <statement> ::= while <expression> do <statement>

whose meaning is given in the macro body.

The restrictions imposed on the type 1 macro permit a more effective macro process; the definition process is outlined in (2.6).

1. A macro definition is recognised against L<sub>E</sub>'s deterministic top down grammar.
2. The target class is scanned and recorded.
3. The macro template is scanned and converted to an internal list representing the new syntactic rule.
4. The macro body is parsed against the target class taking into account the syntactic classes of the formal parameters as they are encountered in the body's text. During this parse a syntax tree is constructed with hooks for the syntax trees supplied at (2.6) call time for the actual parameters.
5. If a nested macro call is encountered in step 4 then the process is temporarily interrupted, the call is evaluated and the resultant (sub-) tree is returned to the outer process as a branch of the outer syntax tree. Thus the final syntax tree representing the parsed macro body corresponds to pure L<sub>B</sub> text.
6. The syntax tree produced in step 4 is linked up to the internal list produced for the template in step 3.

7. The target class, recorded in step 2, and the macro template are checked against  $L_p$ 's top down grammar to ensure that the deterministic condition is not violated.
8. Finally, the template's internal list is attached to the list of alternative rules for the target class and the parser's decision table is updated to reflect the grammar's modification.

A call to a macro may appear in subsequent  $L_p$  text in the syntactic context specified by the macro's target class. We may intuitively see the calling process as a string substitution process like Leavenworth's. For example, we might interpret the following call to the macro defined in (2.4) - appearing in the context of a statement,

(2.7) while  $i < k$  do  $i := i + 1$

as being replaced by the  $L_B$  text

```
(2.8)  begin
        label 11;
        lab 11: if  $i < k$  then
            begin
                 $i := i + 1;$ 
            goto 11
        end
    end
```

In our base language  $L_B$ , labels (eg 11) must be declared in the block in which they are defined.

The calling process for type 1 macros actually proceeds as outlined in (2.9).

1. A macro call is recognised as a compound target reducing from the target class in  $L_p$ 's grammar.
2. The actual parameters are parsed immediately (as sub-targets) and syntax sub-trees are produced.
3. If a nested call is encountered during the parse of an actual  
(2.9) parameter in step 2 then the parsing process is temporarily interrupted, the nested call is evaluated and the resultant



syntax tree is returned to the outer process as a branch of the tree being constructed for the parameter.

4. Copies of the parameters' syntax trees are finally hooked onto a copy of the body's tree for producing a complete syntax tree for the call. This tree may then be traversed for further evaluation to object code.

As a result of the definition process in (2.6) and calling process in (2.9) all  $L_p$  text is scanned and parsed just once. Since type 1 macro bodies are parsed at definition time and actual parameters are parsed at call time, syntax errors may be detected and reported when and where error messages are meaningful. Though we may see the macro process intuitively from Leavenworth's string substitution view our processor is in fact manipulating fully parsed syntax trees.

The macro process we have described so far is nearly equivalent to a macro process proposed by Vidart (1974). He parses macro bodies at definition time and actual parameters at call time. Yet Vidart retains Leavenworth's requirement that templates are prefixed by delimiting tokens, disallows alternative syntaxes in the template and conditional text replacement in the body, and does not attempt any context sensitive type checking.

Building on our process for type 1 macros we shall now seek improvements in these other areas.

#### 2.2.2 Type 2 macros - conditional text replacement

The general form for our type 2 macro is like that for our type 1 macro in (2.3) but its template and body are extended to permit alternative syntaxes and the control of text replacement conditional upon the syntaxes chosen in a macro call.

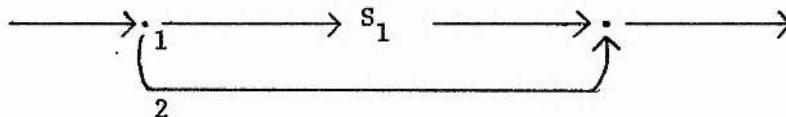
##### 2.2.2.1 Type 2 macro templates

A type 2 macro template is a sequence of quoted basic symbols, formal parameters and/or sub-templates; a sub-template is a syntactic

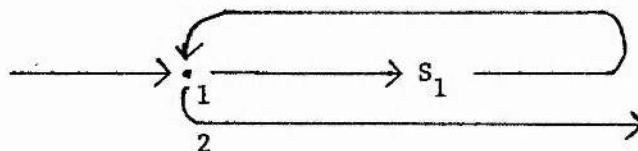


abbreviation on the usual BNF notation and is used for specifying alternative syntaxes in the macro template. Similar to groups in Leavenworth's notation, sub-templates may describe options, lists or alternatives. The three sub-template forms are outlined below with their graphical representations; the subscripted S's each represent a nested template, ie a sequence of quoted basic symbols, formal parameters and/or nested sub-templates.

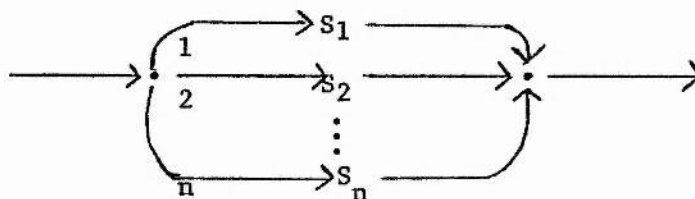
(2.10)  $(? S_1 ?)$  denotes an optional occurrence of  $S_1$ .



(2.11)  $(* S_1 *)$  denotes zero or more occurrences of  $S_1$ .



(2.12)  $(S_1 \mid S_2 \mid \dots \mid S_n)$  denotes a grouping of alternative occurrences  $S_1, S_2, \dots, S_n$  from which one is chosen.



Sub-templates may be assigned names so that they may be referenced in the macro body; this naming of sub-templates takes the form,

(2.13)  $\langle \text{identifier} \rangle : \langle \text{sub-template} \rangle$

For example, consider the following partial macro definition for introducing a new expression form into  $L_p$ .

```

define $expression
(2.14)   rule 'sum' opt: (? 'squares' 'of' ?) $expression1
          others: (* (','| 'and') $expression2 *)
means ...

```

The macro template in (2.14) contains an optional sub-template named "opt", the formal parameter \$expression<sub>1</sub> and a list sub-template named "others"; the sub-template named "others" in turn contains a nested un-named alternative sub-template (for punctuation) and the nested formal parameter \$expression<sub>2</sub>. A macro call to (2.14) might take the form,

(2.15) sum squares of X+1, X-1 and Y.

The graphs in (2.10 - 2.12) indicate the possible paths the parser may follow in recognising the various sub-templates in a macro call. For our underlying L<sub>p</sub> grammar to retain an LL(1) quality the parser must be able to decide which path to take at each decision node based on the next incoming source token. This would require that the sets of possible starting symbols for each path be disjoint.

Our processor in fact relaxes this restriction but retains its deterministic quality by placing an ordering on the paths. If the parser, seeing the next incoming token, can possibly match an option or a list then it does so; similarly, the parser matches the first alternative it can for an alternative sub-template. In the illustrations (2.10 - 2.12) alternative paths in the graphs are numbered in the order in which they are tried. Therefore, in a call to (2.4) such as

(2.16) sum X+1, sum squares of X-1, Y and Z+1

"X+1", "Y", "Z+1" are all actual parameters to the nested macro call.

Of course, we can alter the order of evaluation with parentheses; eg

(2.17) sum X+1, (sum squares of X-1, Y) and Z+1

would have the same effect as

sum X+1, (X-1) \* (X-1) + Y \* Y and Z+1

Another departure we take from LL(1) grammars is a restriction that no template may be matched by the empty string. This is a concession to a more effective implementation whereby modifications to the grammar need not take account of emptiness. This is not as great a restriction as it might appear; the notion of emptiness (or optional forms) can readily be expressed by the use of nested sub-templates. In fact, no syntactic class in the grammar for  $L_B$  (given in Appendix 2) may be matched by an empty string.

#### 2.2.2.2 Type 2 macro bodies

A type 2 macro body may include macro time language ( $L_E$ ) constructs for prescribing text replacement conditional upon which syntaxes in the macro template are matched in a macro call. These  $L_E$  constructs resemble those of Algol in that they have a nested structure but they are non-procedural and are evaluated at call time. Each construct evaluates to a segment of replacement text.

These  $L_E$  constructs are initially best illustrated by example; consider the completed macro definition we began in (2.14).

```

(2.18)  define $expression
         rule 'sum' opt: (? 'squares' 'of' ?) $expression1
           others: (* (','|and) $expression2 *)
         means
         list
         given opt
           then [ ($expression1) * ($expression1) ]
           else [ ($expression1) ],
         forall others:
           given opt
             then [ + ($expression2.others) * ($expression2.others) ]
             else [ + ($expression2.others) ]
         end
       endef

```

In the macro body for (2.18) square brackets bracket segments of replacement text written in the current  $L_p$ ; these segments contain references to formal parameters in the macro template. Formal parameters and sub-templates are referred to by name. Nested parameters and nested sub-templates are referred to as fields of the sub-templates in which they are nested; eg  $\$expression_2.others$  in (2.18) refers to the formal parameter  $\$expression_2$  nested in the sub-template named others. Therefore, the subscripts to  $\$expression$  in (2.18) are not strictly necessary. Two of the three  $L_E$  constructs in the macro body of (2.18) reflect the use of sub-templates in the macro template.

In our descriptions of the  $L_E$  constructs used in the macro body the subscripted T's each represent either a nested  $L_E$  construct or a bracketed segment of replacement text. If  $T_i$  is a bracketed text segment then  $T_i$ 's value is the text itself; the values for the  $L_E$  constructs are given below.

The outermost  $L_E$  construct in the macro body of (2.18) has the general form,

(2.19) list  $T_1, T_2, \dots, T_n$  end

and serves as a bracketing device. Its value is the concatenation of the values for  $T_1, T_2, \dots, T_n$ . In (2.18) this construct brackets two nested  $L_E$  constructs.

The first of these nested constructs has the general form,

(2.20) given <name list> then  $T_1$  else  $T_2$

where each name in the name list is the name of an optional sub-template. The value of (2.20) is  $T_1$  if and only if all optional sub-templates named in the name list are matched in the macro call. We use this construct in (2.18) for determining if the terms being summed are to be squared first.

Another construct in (2.18) has the general form,

(2.21) forall <name list> :  $T_1$

where each name in the list is the name of a list sub-template.

Assuming just one name in the name list initially, the value of (2.21) is the concatenation of successive values of  $T_1$  as  $T_1$  is evaluated for each match of the named list sub-template in the macro call. If there is more than one name in the name list, then each list sub-template named must be matched the same number of times; in this case each evaluation of  $T_1$  is done in the context of each successive match of all the named list sub-templates together. We use this construct in (2.18) for summing (possibly zero) additional terms, matching  $\$expression_2$ , into the initial term, matching  $\$expression_1$ .

Taking all of these  $L_E$  constructs together we may interpret the macro call to (2.18),

(2.22) sum squares of  $X+1$  and sum  $X-1$ ,  $Y$  and  $Z+1$

as producing the  $L_B$  text in (2.23) below.

(2.23)  $(X+1) * (X+1)$

$+ ((X-1) + (Y) + (Z+1)) * ((X-1) + (Y) + (Z+1))$

Note we have made no reference in the macro body of (2.18) to the alternative sub-template nested in "others". This sub-template serves only to provide alternative punctuations, ie ', ' or 'and', in our example; which alternative chosen in a macro call has no semantic meaning. For this reason we do not bother to name the sub-template in the macro template. We now give an example in which an alternative sub-template does have meaning and so is referenced in the macro body.

In our base language  $L_B$ , variables are declared with a let declaration of general form,

(2.24) let <identifier> = <expression>.

Declared variables are initialised; the type of a declared variable is the type of the initialising expression. We might wish to introduce

a declaration form into  $L_B$  which declares a variable to be of a specific type, eg int, bool, str or ptr, but where we do not want to specify an initial value. We can define such a declaration with the following type 2 macro definition.

```

define $declaration
  rule type: ('int'|'bool'|'str'|'ptr') $identifier
means
  choosing type from
  list
(2.25)    [let $identifier = 0],
           [let $identifier = false],
           [let $identifier = ""],
           [let $identifier = null]
  end
endef

```

Here the macro template consists of an alternative sub-template named "type" followed by the formal parameter \$identifier. The macro body makes use of one  $L_E$  construct reflecting the use of the alternative sub-template in the template; this construct has the general form,

(2.26) choosing <name> in list  $T_1, T_2, \dots T_n$  end

where the name refers to an alternative sub-template. The value of (2.26) is  $T_i$  where the  $i^{\text{th}}$  alternative (of the named sub-template ( $S_i$  in (2.12)) is matched in the macro call.

For example, the following calls to (2.25),

(2.27) str s and int i

each would produce

(2.28) let s = "" and let i = 0 respectively.

We have not yet indicated the scope of formal parameter references or sub-template references nested within  $L_E$  constructs. Consider the following type 2 macro definition (assuming the while statement defined in (2.4)).

```

define $statement
  rule 'for' $variable1 ':=' $expression1
    'step': (? step $expression2 ?)
    'to' $expression3 'do' $statement1
  means
    list
      [begin
        $variable1 := $expression1;
(2.29)   while $variable1 <= $expression3 do
          begin
            $statement1;],
        given step
          then [$variable1 := $variable1 + $expression2.step]
          else [$variable1 := $variable1 + 1],
        [end end]
      end
    endef

```

Here we use the given construct for controlling text replacement conditional upon whether or not the step \$expression<sub>2</sub> option is matched in a macro call. It would have been ridiculous to refer to the formal parameter \$expression<sub>2</sub>.step in the else clause since the else clause is evaluated only when the sub-template containing \$expression<sub>2</sub> is not matched at call time. Such contradictions are prohibited in L<sub>E</sub> by a macro time scoping mechanism for formal parameters (or sub-templates) nested within L<sub>E</sub> constructs.

Formal parameters and sub-templates in the outer macro template may be referenced anywhere in the macro body. Formal parameters and sub-templates nested within a sub-template named N may be referenced only,

1. if N is an optional sub-template then within the then clause of a given construct with N in its name list,
2. if N is a list sub-template then within a forall construct with N in its name list, or

3. if N is an alternative sub-template then within the  $i^{\text{th}}$  alternative of a choosing construct with name N where the formal parameter (or sub-template) being referenced lies within the  $i^{\text{th}}$  alternative of N.

With these scope restrictions the processor will never attempt to reference a non-existent actual parameter in the replacement process.

#### 2.2.2.3 A revision to the macro process

The introduction of sub-templates in the macro template and  $L_E$  constructs for conditional text replacement in the macro body introduces differences in the processing of type 2 macro definitions and calls.

Steps 3, 4 and 6 of the definition process described in (2.6) for the type 1 macro must be amended as follows for the type 2 macro.

3. The internal list constructed for representing the macro template is no longer linear but has a nested structure which reflects the template's structure. Portions of this list which represent sub-templates are constructed according to the graphs illustrated in (2.10 - 2.12).
4. The macro time  $L_E$  text in the macro body is translated into (2.30) an abstract program tree which will be executed at call time. References to formal parameters and sub-templates are computed and checked for legal scope. As bracketed  $L_P$  text segments are encountered they are broken into tokens and linked into lists which become leaves of the abstract program tree. No  $L_P$  text will have been parsed at this point.
6. The abstract program tree (not a syntax tree) constructed in step 4 is linked together with the macro template's list produced in step 3.



Steps 2 and 4 of the calling process described in (2.9) for the type 1 macro must be amended as follows for the type 2 macro.

2. Once the start of a call is recognised it is parsed against the macro template's internal list structure. As actual parameters are encountered they are parsed against the formal parameters for producing their syntax trees; as sub-templates are encountered in the template's list we record how they are matched (or not matched). The actual parameters' syntax trees and information on sub-templates are all linked into a call time list structure which reflects the template's list structure;

(2.31) basic symbols are omitted.

4. Macro expansion consists of executing the macro body's abstract program tree with the call time list produced in step 2 as input. The result of execution is a complete segment of tokenised  $L_P$  replacement text containing pointers to the actual parameters' syntax trees. The text segment is parsed against the macro's target class, taking the syntaxes of the actual parameters into account, for producing a complete  $L_B$  syntax tree.

Actual parameters to a type 2 call are scanned and parsed as they are recognised in the call; thus they are parsed just once and syntax errors may be detected and reported where they occur. But the introduction of conditional text replacement makes syntax analysis of type 2 bodies in general impossible at definition time. The parser must deal with  $L_P$  text produced at call time so there is repetitive scanning of text and less effective syntax error diagnosis.

We can partially solve this problem by syntactically tagging bracketed text segments in the body which conform to specific syntactic classes. This tagging has the general form,

(2.32) <syntactic class name> : [ $L_P$  text segment]

where the syntactic class name indicates the expected syntax of the  $L_P$  text segment.

For example we may rewrite the macro body for our definition of a new declaration form in (2.25),

```

... means
      choosing type from
      list
(2.33)  $declaration : [let $identifier = 0],
        $declaration : [let $identifier = false],
        $declaration : [let $identifier = ""],
        $declaration : [let $identifier = null]
      end
endif

```

The parser may then parse tagged text segments at macro definition time; syntax trees replace the text segments as leaves of the body's abstract program tree. Syntax errors occurring within these tagged segments may be diagnosed where they occur at definition time. As more text segments are tagged in the macro body the work required for processing a type 2 macro call approaches that required for a type 1 call; eg since all segments are tagged in (2.33), no parsing of  $L_P$  text is required at macro call time.

### 2.3 Deleting and replacing $L_P$ forms

"Extending"  $L_B$  to  $L_P$  need not be restricted to just adding new linguistic forms to  $L_B$ . For instance, in their paper describing ECT, an extensible-contractible translator, Solntseff and Yezeriski (1972) introduce a device for deleting as well as adding rules to a base language grammar;  $L_P$  may reject unwanted  $L_B$  constructs. We include constructs in  $L_E$  both for deleting and for replacing (or redefining)  $L_P$  forms.

#### 2.3.1 $L_P$ deletions

Our deletion facility has the general form,

(2.34) delete <target class> rule <abbreviated template> endif

The abbreviated template identifies the alternative BNF rule for the target class which is to be deleted in the underlying grammar; a deleted rule is no longer applicable for subsequent macro definitions and  $L_p$  text. The abbreviated template may be either a macro template or a prefix by which the alternative to be deleted can be identified; the prefix may optionally be followed by the character sequence "...".

For example, both deletions in (2.35) effectively remove the if statement from  $L_p$ .

```
(2.35)  delete $statement rule 'if' $expression1 'then' $statement1  
        endif  
        delete $statement rule 'if' ... endif
```

Since type 2 macro definitions appearing before a deletion might use the deleted form in their macro bodies, the processor does not physically remove the internal list representing the deleted form but marks it as no longer applicable to subsequent  $L_p$  text. Marked lists are ignored when subsequent macro definitions are checked against the underlying grammar for LL(1) violations.

### 2.3.2 Replacing $L_p$ forms

Rather than simply deleting a form, one might wish to replace the form using a macro definition for prescribing a template of extended syntax and/or a revised macro body. The general form of a replacement definition with which this may be done is

```
(2.36)  replace <target class> rule <abbreviated template>  
        by <macro template>  
        means <macro body> endif
```

Here the form identified by the macro template replaces the form specified by the abbreviated template for the indicated target class; the new form's meaning is given in the macro body.  $L_p$  text in the macro body may employ the old  $L_p$  form for defining the new meaning.

The new template must have the same starters (tokens, parameters and sub-templates) as the template being replaced.

For example, instead of deleting the if statement as in (2.35) we might redefine it to include an optional else clause with the following replacement definition.

```

(2.37)  replace $statement
        rule 'if' ...
          by 'if' $expression 'then' $statement1
            elsepart: (? 'else' $statement2 ?)
        means
          given elsepart then
            $statement:
              [begin
                label exit;
                if $expression then
                  begin
                    $statement1;
                    goto exit
                  end;
                $statement2.elsepart;
                lab exit
              end]
          else
            $statement: [if $expression then $statement1]
        endef

```

Firstly, we have tagged the two alternative text segments in the macro body as statements; the parser can produce syntax trees for these at definition time. Secondly, we could not have just defined an additional if-then-else statement since that would have conflicted with the original if-then statement in the underlying deterministic grammar; note that we have used the old if-then form for defining the new form.

As with deletions, the  $L_p$  form being replaced in a replacement definition is not physically removed from the underlying grammar but

is marked (internally) as being no longer applicable to subsequent  $L_p$  text.

The macro template in (2.37) does not strictly conform to the LL(1) condition. Though the BNF rule it represents may be distinguished from the other  $L_p$  alternatives for statement (by the 'if'), the LL(1) condition is violated by the presence of the optional else clause. The ambiguity (commonly called the "dangling else problem") arises with nested calls, eg

```
      if Y > 0 then  
(2.38)      if X > 0 then Y := Y+X  
              else Y := Y-X
```

Ignoring the layout, the else clause might refer to either of the two if-then clauses. But our method for matching sub-templates (in section 2.2.2.1) resolves the ambiguity. Since an option is matched whenever possible, the else always refers to the nearest previous if-then; eg (2.38) may be interpreted as it is laid out in the illustration.

## 2.4 Interpretation of the macro process

The advantage of Leavenworth's scheme is that the user may interpret his macros in terms of text replacement; Leavenworth's process precisely reflects this interpretation. At macro definition time the template and body text are copied for representing the macro definition. A macro call, though recognised syntactically, is replaced by the text defining the macro body; the texts of the actual parameters are substituted for formal parameters in this replacement text. Once nested calls have been evaluated, the final  $L_B$  text may be parsed and compiled to object code.

Our syntactic macro processor operates on syntax trees rather than on  $L_p$  text. At macro definition time templates are translated to lists which prescribe the allowable syntaxes for macro calls; type 1

macro bodies (and tagged text segments in type 2 macro bodies) are translated to incomplete  $L_B$  syntax trees with hooks for parameters. At call time the actual parameters of the call are translated to syntax (sub-) trees which are linked into a call time list reflecting the call's syntax; we can think of this list as a sort of syntax tree for the macro call. Macro expansion then involves replacing the call's syntax tree by the syntax tree (or trees in type 2 macros) defining the body; the replacing tree is completed by hooking in the sub-trees for the actual parameters. This completed tree may then be traversed for compilation to object code.

This alteration to the macro process need only concern the processor implementor; the user may continue to interpret his macros in terms of text replacement. This comes from the fact that the syntax trees are only a structural alteration of the  $L_B$  source text; no semantic actions are carried out for their construction. Thus the user gets a more efficient macro process (by less repetitive scanning) and, more importantly, better syntax error diagnosis while being able to retain his textual interpretation of macro expansion.

In the next chapter we shall describe how we can carry the compilation of macro definition bodies and macro call actual parameters even further by executing certain semantic actions on the syntax trees. We shall also examine how such semantic analysis can improve the more context sensitive data type diagnosis and how this must affect one's interpretation of the macro process.

## CHAPTER 3

### Semantic Actions in the Macro Process

In Chapter 2 we described how we can construct syntax trees for macro bodies (or body segments) at definition time and for actual parameters at call time. In this chapter we examine where we may apply semantic functions to these trees, as we construct them, for producing object code trees directly. As we perform more semantic actions at definition time we need to perform fewer semantic actions each time the macro is invoked at call time. Also, if we can check for context sensitive type matching violations in macro bodies at definition time and in actual parameters at call time then type violations, like syntax errors, may be flagged where they appear in the original  $L_P$  source text.

#### 3.1 Computation trees

Hammer (1971) and Cole (1976) have suggested compiling macro bodies directly to object code trees at definition time; such trees might contain hooks, reflecting formal parameters, to which actual parameters are attached at call time. If  $L_B$ 's compilation is syntactically directed we know, in general, what semantic actions are necessary for translating  $L_B$  text segments (or  $L_B$  syntax trees) of any particular  $L_B$  form (specified by a BNF rule) into object code. Given our ability for constructing  $L_B$  syntax trees our problem is to decide when, in the macro process, the semantic actions may be performed. We must distinguish between those actions which we can perform immediately at definition time and those actions which must be deferred to call time when actual parameters are supplied and code is being output. To this end Hammer distinguishes between two sorts of semantic actions: constructive actions and analytic actions.

Constructive actions are those semantic actions in the compilation process concerned with the physical generation of object code. The object code generated is often a constant of the syntax for the linguistic form being compiled without regard to the syntaxes or meanings of its sub-forms. This is best illustrated by example; reconsider the macro body for the type 1 macro we defined in (2.4) for introducing a while statement.

```
(3.1)  [begin
        label l1;
        lab l1: if $expression1 then
            begin
                $statement1;
                goto l1
            end
        end]
```

We can outline the object code produced for this body at definition time; assuming a target stack machine we might translate (3.1) to the following code segment.

```

i : * (code for $expression1)
    JUMPF j
(3.2)  * (code for $statement1)
    JUMP i
j :
```

Asterisks (\*) indicate hooks to which code segments (trees) for actual parameters will be attached when the macro is invoked at call time. The remaining code will be generated for every call to the macro. The "JUMP i" instruction indicates an unconditional branch to the label i. The "JUMPF j" indicates a branch to j on the condition that the boolean value atop the run time machine stack (computed by the code for \$expression<sub>1</sub>) is false; the value is popped from the stack whether true or false. Since the macro may be invoked several times we must create unique labels for i and j each



time the code segment (3.2) is actually output as a portion of the final object deck. But we at least know at definition time the pattern of branching required in the code segment. As we are constructing syntax trees throughout the macro process we might as well take advantage of this characteristic of constructive actions and execute as many as we can for producing partial code trees.

On the other hand, analytic actions perform the more analytic chores required in the compilation process such as looking up names, mapping names to addresses and type checking. Analytic actions often impose certain context sensitive restrictions (eg types) on  $L_B$  which cannot be expressed in the underlying context free grammar. The execution of such actions on  $L_B$  syntax trees generally requires information about the trees' constituents and the surrounding context; in the macro process such actions must be deferred to call time. An analytic action required in the compilation of (3.1) is a check that the value computed for  $\$expression_1$  is of boolean type; this would necessarily have to be checked at macro call time.

Some constructive actions such as the generation of object code for generic operators require the execution of analytic actions; eg the code produced for relational operators "=" and " $\neg$ =" might depend on the operand types. We can treat such constructive actions as analytic actions and defer them to call time. Other actions, like the creation of unique labels and the mapping of declared names to stack addresses must be deferred not just until call time but until code is actually emitted in the compilation of an  $L_P$  program.

Hammer suggests we produce computation trees for macro bodies (he considers only type 1 macros) at definition time; a computation tree is essentially a syntax tree for which constructive semantic actions have been executed. Alongside each tree Hammer would construct a 'deferred list' of analytic actions which must be executed each

time the macro is invoked. A subsequent macro call is then replaced by this computation tree, actual parameter trees are substituted for formal parameters and the actions in the deferred list are executed for performing the more analytic chores.

In our scheme we shall construct computation trees in a manner similar to that we described in Chapter 2 for constructing syntax trees. As we parse a macro body segment at definition time or an actual parameter at call time we execute all possible constructive actions; at the same time we embed calls to routines which perform analytic actions in the computation tree being constructed. Our computation tree will contain object code, hooks for actual parameter trees and calls to those routines which will perform the necessary analytic chores when invoked at macro expansion time.

For example, a (simplified) computation tree for (3.1) might take the form,

```
      i : * (a hook for $expression1's code)
          poptype (bool)
(3.3)  JUMPF j
          * (a hook for $statement1's code)
          JUMP i
      j :
```

where asterisks indicate hooks for actual parameter computation trees, uppercase words indicate object code instructions and lowercase words indicate calls to deferred actions which cannot be executed until call time. The deferred routine call 'poptype (bool)' would, at call time, check that the expression given as an actual parameter for \$expression<sub>1</sub> is of boolean type. The computation tree in (3.3) would necessarily be more complicated by additional deferred routine calls for computing unique addresses for labels i and j as code is emitted.

In subsequent macro calls a computation tree is constructed for each of the actual parameters. For example, consider the

following call to our while macro.

```
      while x <= 16 do  
        begin  
(3.4)    writeln x;  
          x := x+1  
        end
```

For the expression, 'x <= 16', we might construct the computation tree,

```
      loadaddr(x)  
      CONTENTS  
(3.5) LOADCONSTANT 16  
      stacktype(int)  
      relation(<=)  
      stacktype(bool)
```

and for the block statement (bracketed by begin and end) the tree,

```
      loadaddr(x)  
      CONTENTS  
      poptype(int)  
      WRITEINT  
  
      loadaddr(x)  
(3.6) loadaddr(x)  
      CONTENTS  
      type(int)  
      LOADCONSTANT 1  
      ADD  
      checktypes  
      STORE
```

The 'loadaddr(x)' call produces code for loading the address of x onto the run time stack and places x's type atop the compile time stack. The compile time stack is used for computing, at compile time, the types of those expressions to be computed atop the target machine's evaluation stack at run time. The routine 'stacktype(T)' will place type T atop the compile time stack, 'poptype(T)' checks that T matches the type popped off that stack, 'type(T)' checks type T against the stack's top type (without popping it off) and 'checktypes'

pops the two top types from the stack and checks that they match. The routine call, 'relation(<=)', is used for handling the generic relational operator (<= in (3.4)); it pops the two top types from the compile time stack, checks that those types match and then produces object code for computing the boolean result of the relational test. The meanings of the (uppercase) object code instructions in (3.5) and (3.6) should be obvious from their contexts.

Finally, (copies of) the computation trees constructed for the actual parameters, eg (3.5) and (3.6), are hooked into a copy of the computation tree for the macro body, eg (3.3); the deferred routine calls are evaluated for checking types and producing object code for our target stack machine. Thus macro expansion of the call (3.4) might produce the following code segment.

```
      i' : LOADADDR x'
          CONTENTS
          LOADCONSTANT 16
          LE
          JUMPF j'
(3.7)  LOADADDR x'
          LOADADDR x'
          CONTENTS
          LOADCONSTANT 1
          ADD
          STORE
      j' :
```

Here i' and j' represent unique object program addresses and the x' represents the run time stack address of the simple variable x.

The important point is that computation trees may be constructed for macro body segments at definition time and for actual parameters recognised at call time; thus computation trees need be constructed just once for each  $L_p$  text segment. Since computation trees are constants of those syntax trees to which they correspond then, like

syntax trees, their construction will not corrupt the user's text replacement interpretation of the syntactic macro process. But, while this construction of computation trees improves the efficiency of the macro process, it gives nothing more than the construction of syntax trees gives in terms of more effective error diagnostics. That is, the construction of computation trees has not improved the more context sensitive type diagnostics as the construction of syntax trees, in Chapter 2, improved the context free syntax diagnostics.

In the remaining sections of this chapter we shall examine where we can execute some of the (normally deferred) analytic actions immediately as we construct our computation trees. This should not only allow the diagnosis of type violations where they occur in the source text but should also modify the way in which the user must interpret macro expansion. We shall first treat the computation trees for the actual parameters of a macro call; this will set the stage for our treating trees for macro bodies at definition time.

### 3.2 Diagnosing type violations in actual parameters

Suppose that in constructing a computation tree for an actual parameter to a macro call we were to execute its (normally deferred) analytic actions immediately. Firstly, this would permit the diagnosis of type violations internal to the parameter as it is scanned; such violations could be flagged just where they appear in the parameter's  $L_p$  text. Secondly, for those actual parameters which have types (eg parameters matching \$expression, \$term or \$name), we can compute each parameter's type atop the compile time type stack.

For example, consider the case where the macro call (3.4) appears in a  $L_P$  program with variable  $x$  declared globally to that call<sup>1</sup>,

```

      let x = 0;
      :
      while x <= 16 do
        begin
(3.8)   writei x;
          x := x+1
        end;
      :

```

As we scan the call's first parameter ' $x \leq 16$ ' and construct its computation tree in (3.5), each time we would normally include a deferred routine call (eg ' $\text{loadaddr}(x)$ ' in (3.5)) we instead execute the 'deferred' routine directly. In this way we perform type checking and generate pure object code immediately as we construct our computation tree.

Since  $x$  has an integer type (it takes the type of its initialising value), this process would verify the parameter's legality, leave the parameter's type (bool) atop the compile time type stack and produce the following computation tree.

```

      LOADADDR x'
(3.9) CONTENTS
      LOADCONSTANT 16
      LE

```

Having constructed (3.9) we might pop the type (bool) from the compile time type stack and save it (along with the tree) for later use.

We could similarly construct a pure code tree for the second (block statement) parameter; since statements have no types, no type would be computed atop the compile time stack.

---

1 For now it is important that the call is not nested within the body of another macro definition; if it were we could not be sure of the type declared for  $x$ . We discuss nested calls in section 3.3.3.

Thus type violations, like syntax errors, occurring in an actual parameter may be detected and flagged when they are first scanned rather than after the call has been fully 'expanded'. For example, if x had been of string type then our compiler/processor would be able to flag the mismatching of types for the '<=' operator just where the mismatch appears in the source listing;

```
    let x = "a string value";  
    :  
(3.10 while x <= 16 do  
    *** ERROR :mismatched types ('str' and 'int')  
    begin  
        writeln x;  
        x := x+1  
    end  
    :
```

We should also like to be able to specify expected types (where applicable) for the actual parameters themselves. In terms of our constructing computation trees this means we should like to specify, at definition time, that type we expect to be computed for an actual parameter atop the compile time type stack at call time. We may prescribe such type restrictions on actual parameters in terms of their corresponding formal parameters in the macro definition. This will allow the reporting of any type violations for an actual parameter in terms of the parameter's use in the macro call itself rather than in terms of its use in the macro body after macro expansion. We propose two mechanisms for this purpose below; the mechanism chosen will depend on the flexibility required for prescribing the type restrictions.

### 3.2.1 Type restrictions specified in the template

A simple method for restricting any actual parameter matching a formal parameter to be of a specific type is to tag the formal parameter with the expected type in the macro template. The general form

used in tagging a formal parameter is

(3.11) <formal parameter>.<expected type>

where the expected type is that type to which all actual parameters matching the formal parameter must conform.

For example, we may restrict the conditional expressions matching  $\$expression_1$  in calls to our while macro (2.4) by rewriting the macro definition as follows.

```

define $statement
  rule 'while' $expression1. bool 'do' $statement
means
  [begin
    label 11;
(3.12)    lab 11: if $expression1 then
            begin
              $statement1;
            goto 11
            end
          end]
  endef

```

At call time, any actual parameter matching  $\$expression_1$  (syntactically) would then be checked immediately for boolean type. This could be done by checking that type computed atop the compile time type stack once we have constructed a computation tree (and executed the analytic routines) for the actual parameter. If the actual parameter matching  $\$expression_1$  in (3.12) were not of boolean type then the violation could be flagged just where it occurs in the source listing; eg

```

while X+Y do
(3.13) *** BAD TYPE : 'bool' expected where 'int' found.
begin
  :

```

This tagging of formal parameters in the macro template may be done for type 2 macros so long as each parameter is to be restricted to a specific constant type. For example, we could tag some of the



parameters in the template for our for statement macro in (2.29) as follows.

```

      define $statement
      rule 'for' $var1. int '[:=' $expression1. int
(3.14)      step : (? 'step' $expression2. int ?)
            'to' $expression3. int 'do' $statement1
      means ...

```

At call time, any actual parameters matching \$var<sub>1</sub>, \$expression<sub>1</sub> or \$expression<sub>3</sub> in (3.14) must be of integer type. For calls having a step expression, matching the optional sub-template named step, the actual parameter matching \$expression<sub>2</sub>.step must also be of integer type.

We need not restrict ourselves to checking simple types. L<sub>B</sub> has a declaration form for defining structured types; eg the declaration,

```
(3.15) structure list (int info, ptr next)
```

declares "list" to be a constructor name, whose type is written "(int,ptr)" and which may be used for constructing structured objects. For example, the primary form

```
(3.16) construct list (5,null)
```

returns a pointer to a structured object with an integer field with value 5 and a pointer field with value null - the special value of type ptr.

We may check the types of constructor names passed as actual parameters to the macros we define. For example, the following type 1 macro definition introduces a new primary form to L<sub>p</sub> of type ptr.

```

      define $factor
      rule 'a' $name. (int,ptr) 'node' $expression1. int
(3.17)      'pointing' 'to' $expression2. ptr
      means
            [construct $name ($expression1, $expression2)]
      endef

```

Since all parameters in the template have been restricted to specific types, the macro body may be parsed and diagnosed for type violations at definition time.

Given the definition (3.17), the declaration (3.15) and a variable  $p$  of type ptr, the effect of the  $L_p$  statement,

(3.18)  $p := \underline{a \text{ list node } 5 \text{ pointing to } p}$

would be to point  $p$  to a structured "list" object. The field  $\text{info}(p)$  would have the value 5 and the field  $\text{next}(p)$  would point to  $p$  itself.

Tagging a formal parameter in a macro template for denoting a specific expected type is analogous to typing the parameters of procedures in many high level programming languages (eg Algol 68). In defining syntactic macro extensions we often require greater flexibility for specifying type restrictions, for instance where we require that two parameters have equal types. For this purpose we have defined a where clause for use in conjunction with macro definitions.

### 3.2.2 The where clause

Consider the following (segment of a) macro definition which may be used for replacing the simple  $L_B$  assignment statement by a new construct allowing multiple assignments.

```

replace $statement
  rule $var ...
(3.19)   by $var1 vars : (* ',' $var1 *) ':= '
          $expression1 exprs : (* ',' $expression1 *)
  means ...

```

A macro call to (3.19) takes the form of a list of variables separated by commas, followed by the ':= ' operator, followed by a list of expressions separated by commas; eg

(3.20)  $X, Y, Z := \text{"a string"}, 17, Z+5$

We should like (3.20) to have the effect that X takes on the string value "a string", Y takes on the integer value 17 and Z takes on the integer value of Z+5. We should like to restrict calls to (3.19) in general such that the lists of variables and expressions are of equal length and corresponding variables and expressions have like types; eg X must be a string variable and Y and Z must be integer variables. Such restrictions may be prescribed in the macro definition using the where clause.

The general form for a macro definition containing a where clause is

```
define <target class> rule <macro template>  
  where <assertion >  
  means <macro body> endif
```

(3.21) (or)

```
replace <target class> rule <abbreviated template> by <macro template>  
  where <assertion >  
  means <macro body> endif
```

where assertion is a (possibly compound) assertion which must be satisfied by the actual parameters of the call matching the macro template.

For example we may complete the macro definition (3.19) as follows.

```

replace $statement
  rule $var ...
    by $var1. eval vars : (* ',' $var1.eval *) ':= '
      $expression1. eval exprs : (* ',' $expression1.eval *)
  where
    list
      eqlen (vars,exprs),
      match (type $var1, type $expression1),
      forall vars,exprs:
(3.22)      match (type $var1.vars, type $expression1.exprs)
    end
  means
    list
      [begin $var1 := $expression1],
      forall vars,exprs :
        [; $var1.vars := $expression1.exprs],
      [end]
    end
  endef

```

The process for handling a macro call to (3.22) may be broadly divided into three steps:

1. (Recognition) Firstly, the call is recognised against the macro template and computation trees are constructed for actual parameters. The construction of trees for actual parameters matching formal parameters tagged with eval in the template includes the immediate execution of analytic actions for computing each parameter's type.
2. (Verification) Secondly, the (compound) assertion in the where clause is evaluated for verification.
3. (Expansion) Finally, if the macro call and its parameters satisfy the assertion in 2 then the macro call is expanded as prescribed in the macro body; otherwise an error message is given.

The assertion in the where clause of (3.22) is a compound list

of three assertions each of which must be satisfied for the compound to be satisfied. Of these three nested assertions, the first two are primitive assertions, or predicates, and the third is an assertion made for a list of matches to a list sub-template.

Primitive assertions are predicates referring to the actual parameters to a call by way of names in the macro template. For example, the predicate

(3.23) match (<type designator>,<type designator>)

has the value true if and only if the two designated types are equal. A type designator in (3.23) may be either a constant (eg int or bool) or of the form,

(3.24) type <formal parameter ref>

referring to the type of the actual parameter matching the specified formal parameter. Another predicate in (3.22) has general form,

(3.25) eqlen (<length>,<length>)

where length is either an integer constant or the name of a list sub-template; in the latter case the value is the number of times (possibly 0) the list subtemplate is matched in the macro call.

The predicate (3.25) is true if and only if the two lengths are equal. We also have the following compound predicates with obvious values.

not (<predicate>)

(3.26) or (<predicate>,<predicate>)

and (<predicate>,<predicate>)

Compound assertions are expressed using  $L_E$  constructs which are orthogonal to the  $L_E$  constructs introduced in section 2.2.2 for specifying conditional expansion in the macro body.  $L_E$  constructs in the where clause are used for making assertions in the context of sub-templates. In our discussion of these  $L_E$  constructs below, subscripted A's denote nested primitive or compound assertions.

The list construct serves to bracket several assertions and

has the general form

(3.27) list A<sub>1</sub>, A<sub>2</sub>, ..., A<sub>n</sub> end

The assertion (3.27) is satisfied if and only if all nested assertions A<sub>1</sub>, A<sub>2</sub>, ..., A<sub>n</sub> are satisfied.

The forall construct used in the where clause of (3.22) has the general form

(3.28) forall <name list> : A<sub>1</sub>

where each name in the name list is the name of a list subtemplate. For (3.28) to be satisfied, A<sub>1</sub> must be satisfied for each successive match of the named subtemplates. For example, in (3.22) the first variable matched for vars must have the same type as the first expression matched for exprs, the second variable matched for vars must have the same type as the second expression for exprs, and so on. Of course, the list subtemplates vars and exprs must be matched an equal number of times.

Given (3.23), (3.25), (3.27) and (3.28) we may now interpret the where clause in the macro definition in (3.22). Any call to (3.22) must have an equal number of variables and expressions; the first variable must have the same type as the first expression and all subsequent variables must have the same types as corresponding expressions.

We also have L<sub>E</sub> constructs for making compound assertions for parameters matched by optional and alternative subtemplates. The first of these has the general form

(3.29) given <name list> then A<sub>1</sub> else A<sub>2</sub>

where each name in the name list names an optional subtemplate.

If all named subtemplates have been matched in the macro call then (3.29) is satisfied if A<sub>1</sub> is satisfied; otherwise (3.29) depends on A<sub>2</sub> being satisfied. The second L<sub>E</sub> construct has general form

(3.30) choosing <name> in list A<sub>1</sub>, A<sub>2</sub>, ..., A<sub>n</sub> end

where the name denotes an alternative subtemplate. The assertion

(3.30) is satisfied if and only if  $A_i$  is satisfied where the  $i^{\text{th}}$  alternative in the subtemplate was matched at call time. Like the  $L_E$  constructs in the macro body, the  $L_E$  constructs above impose scope rules on their nested assertions. This prevents the user from making assertions about actual parameters not recognised in a macro call.

There is an additional  $L_E$  construct particular to assertions in the where clause. Since assertions are not verified until the entire call has been recognised and since these assertions are user defined, the user must have some means of producing meaningful error messages when the macro call does not satisfy his assertions. He may define such error messages with the following  $L_E$  construct.

(3.31) assert  $A_1$  :<string>

The message specified in the string is emitted just after macro calls for which  $A_1$  is not satisfied. The construct (3.31) is itself an assertion which is satisfied when  $A_1$  is satisfied. For example, we may rewrite the where clause in (3.22) as follows for providing meaningful error messages.

```

where
  list
    assert eqlen (vars,exprs) : "unbalanced assignment",
    assert match (type $var1, type $expression1):
(3.32)      "mismatched types for assignment",
    forall vars,exprs:
      assert
        match (type $var1.vars, type $expression1.exprs):
          "mismatched types for assignment"
  end

```

It is important that we distinguish between the tagging of parameters in the macro template and making assertions in a separate where clause. Wherever possible, it is best that parameters are restricted in the first way since violations may be detected and reported just where they occur. For example, we might have restricted

the conditional expression parameter to the while macro in (3.12) with the where clause,

```
(3.33) where assert match (type $expression1, bool):  
      "bad type for conditional expression"
```

In this case the user defined message would not be emitted in the source listing until after the entire call; eg

```
      while X+Y do  
      begin  
(3.34)      :  
      end  
      *** USER DEFINED ERROR: "bad type for conditional expression"
```

The error message in (3.34) does not compare favourably with that in (3.13). Of course, the where clause must be employed when we are not restricting parameters to specific constant types.

Since the  $L_E$  constructs for making assertions in the where clause are similar to the  $L_E$  constructs for prescribing text replacement in the macro body one might be tempted to combine the two functions together. The author proposed such a scheme where context sensitive error messages could be conditionally emitted during text replacement (Campbell, 1978). But the specification of context sensitive assertions on a macro call and the specification of conditional text replacement are separate functions. We may think of an assertion as a condition that a macro call satisfies certain restrictions but think of a macro body as a piece of  $L_B$  text. For this reason we have followed Schuman and Jorrand (1970) in keeping the two functions separate in the macro definition.

Whether we tag parameters in the macro template or query parameters in the where clause, the execution of the (normally deferred) analytic actions as computation trees are constructed for the actual parameters has an effect on the user's string replacement interpretation of macro expansion. We now discuss this modification before discussing type diagnosis in the macro body.



### 3.2.3 Effects on the user's interpretation

Type diagnosis in the actual parameters involves the execution of analytic actions as the parameters are recognised in the macro call. Names are evaluated at this point before the actual parameters have been substituted for formal parameters in the macro body. The early evaluation of names in the actual parameters forces one to modify his usual textual interpretation of parameter substitution.

For example, suppose we were to redefine the for macro in (2.29) so that the control variable is declared locally to the macro call; eg

```

define $statement
  rule 'for' $identifier1 ':' $expression1 'to' $expression2
    'do' $statement1
  means
    [begin
      label l1;
      let $identifier1 = $expression1;
      lab l1: if $identifier1 <= $expression2 then
        begin
          $statement1;
          $identifier := $identifier + 1;
          goto l1
        end
      end]
    endef

```

(3.35)

Consider the following  $L_P$  text segment where the control variable to a call to (3.35) is declared globally to the call.

```

let i = 6;
(3.36)  :
        for i := 1 to 5 do write i

```

Our usual textual interpretation of an expansion of (3.36) would be

```

let i = 6;
  ⋮
  begin
    label 11;
    let i = 1;
(3.37)  lab 11: if i <= 5 then
      begin
        writeln i;
        i := i+1;
        goto 11
      end
    end

```

This is just what we might expect; the code generated for (3.37) would at run time write out the sequence '1 2 3 4 5'. But let us suppose that all analytic actions were executed immediately as computation trees were built for the actual parameters to the call in (3.36). Then the variable *i* in the parameter 'writeln i' would be bound at call time to refer to the *i* declared globally; the textual representation of this would be

```

let i = 6;
  ⋮
  begin
    label 11;
    let i' = 1;
(3.38)  lab 11: if i' <= 5 then
      begin
        writeln i;
        i' := i'+1;
        goto 11
      end
    end

```

where *i'* is different from *i*. This is not what we want since the code generated for (3.38) would at run time write out the sequence '6 6 6 6 6'.

Since the identifiers, in those actual parameters which are

typed diagnosed at recognition time, are bound before substitution into the macro body we must view the parameters as being passed 'by reference' rather than by textual substitution. Our string replacement interpretation is corrupted whenever identifiers passed as parameters are rebound by declarations in the macro body, eg as in (3.35).

For this reason we propose that actual parameters are, by default, evaluated only to the computation trees discussed in section 3.1; in this way no analytic actions are performed, no type diagnosis is done and the passing of parameters may be interpreted textually. Should the user wish type diagnosis performed then he must explicitly tag corresponding formal parameters in the macro template. This may involve the tagging of parameters for restricting them to specific types as in section 3.2.1 or with the reserved word 'eval'; all parameters referred to in a where clause must be tagged with 'eval' in the template. Those actual parameters matching tagged formal parameters are checked for type violations, translated to pure code trees and passed by reference. Those matching untagged formal parameters are passed textually.

For example, we can tag some of the formal parameters in the template of our for macro definition in (3.35) as follows.

```
define $statement
  rule 'for' $identifier1 ':= ' $expression1.int
(3.39)      'to' $expression2.int
           'do' $statement1
  means ...
```

In a macro call to (3.39) the actual parameters matching \$expression<sub>1</sub> and \$expression<sub>2</sub> would be checked for internal type violations, checked as having integer types and passed by reference; those matching \$identifier<sub>1</sub> and \$statement<sub>1</sub> would be passed textually. Our selective tagging here preserves the meaning we wish calls to

(3.39) to have as illustrated in (3.37).

Therefore we have two processes for constructing computation trees for actual parameters to a macro call. An actual parameter passed textually is processed in the way described in section 3.1; its computation tree contains calls to analytic routines which are to be executed after the tree is substituted into the tree for the macro body. For an actual parameter passed by reference, analytic routines are executed immediately for binding identifiers, checking types and producing a computation tree of pure object code.

Our requirement that the user tags formal parameters in the macro template ensures that he is aware of which parameters are to be passed by reference and which are to be passed textually. As he tags more formal parameters at definition time, more actual parameters can be effectively diagnosed for type violations at call time.

### 3.3 Diagnosing type violations in a macro body

We have indicated (in section 3.1) how we can construct computation trees at macro definition time for type 1 macro bodies and for syntactically tagged text segments in type 2 macro bodies; such trees contain calls to analytic routines which are executed at macro expansion time for checking types and producing pure object code. We shall now see where we can take advantage of those type restrictions placed on the formal parameters in the template and directly execute some of the analytic routines for diagnosing type violations in the macro body at macro definition time.

#### 3.3.1 Type violations in a type 1 macro body

Consider a version of our while macro definition where all formal parameters have been tagged in the template, the first parameter being restricted to boolean type.

```

define $statement
  rule 'while' $expression1.bool 'do' $statement1.eval
means
  [begin label l1;
    lab l1: if $expression1 then
(3.40)      begin
              $statement1;
              goto l1
            end
          end]
  endef

```

We know, at definition time, that any actual parameter substituted for  $\$expression_1$  in the body of (3.40) at call time will have boolean type. It is therefore unnecessary to call 'popcheck(bool)' in (3.3) for checking its type for use as a condition in the if statement each time the macro is expanded. We can in fact perform all type checking and label verification for the body in (3.40) at definition time and produce the following code tree.

```

      i : * (hook for actual parameter matching $expression1)
          JUMPF j
(3.41)  * (hook for actual parameter matching $statement1)
          JUMP i
      j :

```

It would appear we could generalise this process to all type 1 macro definitions whose parameters having types are restricted to specific types in the macro template. Yet there is an additional complication concerning the binding of names in the macro body which requires our imposing one further restriction.

If a type 1 macro template contains the formal parameter  $\$declaration_1$  then the declaration passed as an actual parameter could have binding effects on the names in a macro body at expansion time which are unpredictable at definition time. In order to perform type diagnosis in a macro body at definition time we must require that all name binding in the body is fixed. We might just prohibit

\$declaration parameters in those type 1 macros for which we require definition time type diagnostics. But since such a restriction would fix the declaration (and so, the binding) of names in the body then we may just as well require that all parameters be passed by reference. In short, when the user desires type checking in his type 1 macro body at definition time, he must tag all formal parameters in the template; those parameters not having types must be tagged with 'eval', eg \$statement<sub>1</sub> in (3.40).

Declarations passed as parameters by reference, ie matching \$declaration.eval, apply only in the call itself, not in the expanded body text after substitution, and therefore have no practical use. Declarations passed as parameters on their own for use in the body must be passed textually; in this case no type diagnosis may be done in the body at definition time.

This restriction does not apply to declarations nested in other sorts of actual parameters (eg in a block statement matching \$statement<sub>1</sub> in (3.40)) since such declarations are local to the parameters by the nature of  $L_B$ . Nor must we prohibit explicit declarations in the type 1 macro body so long as they are not passed as parameters; these may be taken account of at definition time for diagnosing type violations.

For example, in the following type 1 macro definition for introducing the factorial operator we can check types in and produce pure code for the macro body at definition time.

```
define $factor
  rule 'factorial' $factor1.int
means
  [begin
    let i = $factor1;
    let fact = 1;
(3.42) while i >= 2 do
    begin
      fact := fact * i;
      i := i - 1
    end
    → fact
  end]
endef
```

Any type violations we might have introduced in defining (3.42) could then be detected and reported where they occur in the source listing provided for (3.42).

Since not all type 1 macro definitions may be processed in this way we shall require that the user explicitly tags those bodies for which he requires definition time type diagnosis. This tagging has the general form,

```
(3.43) ... means eval : [<LP text>] endef
```

Type 1 macros whose bodies are so tagged must therefore satisfy the following two conditions.

1. Each formal parameter which may have a type must be restricted to a specific type in the macro template.
2. Those parameters not having types must be tagged with 'eval' in the macro template.

All such parameters are passed by reference.

### 3.3.2 Type violations in a type 2 macro body

Unfortunately, it is in general impossible to diagnose type violations in a type 2 macro body at macro definition time, even if all parameters have been restricted to specific types in the macro

template. Conditional expansion permits declarations in the body which may or may not be expanded at call time; this means the type of a variable can vary from one call to the next. This is illustrated in the following type 2 macro definition fragment.

```

means
  :
  given option
    then $declaration : [let X=0]
(3.44) else $declaration : [let X = "a string value"],
    :
    $statement : [X := X+1]
    :
endef

```

The fragment is not unnatural assuming the statement segment 'X := X+1' is nested within an  $L_E$  control construct and is therefore conditionally expanded at call time. All text segments in (3.44) are tagged with their syntaxes and can be parsed and translated to computation trees (containing calls to deferred analytic routines) at macro definition time. But we cannot execute the analytic routines for checking types in 'X := X+1' before the given construct has been evaluated at call time for declaring X to be of either integer type or string type.

In general, a type 2 macro body must be expanded for a particular call before type diagnostic routines can be effectively applied; therefore, any error messages must follow a call to the macro in the source listing even if they apply to type violations in the body of the macro definition. The best we can do for the user in this situation is to indicate that such messages apply to the expanded macro as opposed to the parameters of the call; eg the following error message might follow a call to (3.44).

```

(3.45) *** ERROR NESTED IN EXPANDED CALL : BAD TYPES -
        int expected where str found.

```



This does not mean that the user should be discouraged from defining type 2 macros.

### 3.3.3 Nested macro calls

It is possible to have the situation where a macro call whose actual parameters are restricted to specific types is nested within a macro body for which we are not checking types at definition time. For example, the defining body in the following macro definition employs a nested call to the while macro defined in (3.40).

```
define $statement
  rule 'for' $identifier1 ':= ' $expression1.int
      'to' $expression2.int
      'do' $statement1
means
  [begin
(3.46)    let $identifier1 = $expression1;
          while $identifier1 <= $expression2 do
          begin
            $statement1;
            $identifier1 := $identifier1+1
          end
        end]
endef
```

For reasons discussed in section 3.2.3 we do not want to check types and so bind names in the body of (3.46) at macro definition time. But our definition of the while macro in (3.40) requires that its actual parameters are checked for types as they are recognised at call time; what is call time for the nested while call is definition time for (3.46) as a whole.

We find a similar problem when a call whose actual parameters are restricted to specific types is nested within an actual parameter for which types are not to be checked at recognition time, eg when a call to the while macro is nested within an actual parameter for \$statement<sub>1</sub> (3.46). In either case we are asking that analytic

semantic actions be executed for checking types on a particular branch of a computation tree where types have not been computed on the surrounding parent tree; this makes nonsense of the context sensitivity of types.

But prohibiting such nesting of macro calls would be unnecessarily restrictive and would discourage the  $L_E$  user from restricting parameters to specific types in his macro templates. We instead modify our macro process so that it does no type checking for calls nested within text segments which are not themselves being checked for types. For example, in the construction of a computation tree for the defining body of (3.46), the nested call to the while macro is expanded to a computation sub-tree which (like the surrounding tree for the entire body) contains calls to analytic semantic routines which are to be executed later.

This means the user will not get the effective reporting of type violations in the actual parameters to the while call nested in (3.46) which he would normally get elsewhere; a bad (non-boolean) type for its expression parameter cannot be detected, nor reported in the source listing, until after a call to the for macro itself. On the other hand, the nested call to the while macro in the defining body of our factorial macro in (3.42) permits immediate type checking of the call's actual parameters since the surrounding body may (if tagged with eval) be type diagnosed at macro definition time.

In general, as our processor translates segments of text for which no type checking is prescribed, eg in actual parameters passed textually or in type 2 macro body segments, all calls nested in these segments are likewise translated without any type checking. The user need not be aware of this when he places type restrictions on parameters in his macro definitions; but the processor takes advantage of the user's type restrictions only where type checking is meaningful.

Of course, the actual parameters to any macro call appearing at the outermost level, ie in an  $L_p$  program, may be checked for type violations directly.

### 3.3.4 Computing addresses in the macro process

As we have previously noted in section 3.1, certain semantic actions such as the computation of unique (object code) addresses for labels cannot be executed until we emit code as output from our processor/compiler; this also holds true for the (runtime stack) addresses for variables. Although we may execute certain analytic semantic actions for checking types while we construct computation trees for macro bodies and actual parameters, those semantic actions which compute unique runtime addresses must be deferred to that time that the code trees are actually about; we call this time code generation time.

For example, consider the following type 1 macro which introduces our factorial operator.

```

$ factor
define $factor
  rule 'factorial' $factor1.int
means
  eval: [begin
(3.47)      let fact = 1;
            for X := 2 to $factor1.do
              fact := fact * X
            → fact
            end]
  endef

```

The macro body for (3.47) has been tagged for type diagnosis at macro definition time and the macro's only parameter,  $\$factor_1$ , has been restricted to integer type in the template. Note the nested call to our for macro whose definition in (3.46) in turn employs a nested call to our while macro defined in (3.40). Although this nested call to the for macro cannot be checked for types as it is recognised, the analytic semantic actions on its computation tree

can be executed for checking types in the context of the tree constructed for the entire definition (3.47). So we may produce the following 'pure code' tree for (3.47) at macro definition time.

```

LOADCONSTANT 1 (atop stack as 'fact')
LOADCONSTANT 2 (atop stack as 'X')
i: LOADADDR    X
  CONTENTS
    * (actual parameter code tree for $primary1)
    LE
      JUMPF      j
      LOADADDR   fact
      LOADADDR   fact
      CONTENTS
      LOADADDR   X
(3.48) CONTENTS
      MULT
      STORE
      LOADADDR   X
      LOADADDR   X
      CONTENTS
      LOADCONSTANT 1
      ADD
      STORE
      JUMP      i
j: LOADADDR   fact
  CONTENTS

```

} fact := fact \* X

} X := X+1  
(in for loop of (3.42))

} → fact

Since the code tree in (3.48) is not to be output immediately, but saved as the definition for (3.47) we cannot compute unique runtime addresses for labels i and j and variables fact and X. We must compute these addresses each time (3.48) is emitted as part of the final object code deck; the addresses computed will then be unique to each appearance of (3.48) in that final deck.

Therefore, in constructing a 'pure code' computation tree for a macro body like that in (3.47), we simulate the declarations of labels and variables and then check types against the simulated

declarations. We include, in the computation tree (eg in (3.48)), calls to semantic routines which are invoked at code generation time for computing unique runtime addresses.

We process actual parameters in a similar way. Any labels or variables declared locally (in the source text) to an actual parameter are simulated by the processor as it constructs the parameter's computation tree. Actual addresses are not computed for the parameter's code tree until that tree is output at code generation time.

### 3.4 Generic macros

Since we can check the types of macro parameters we can also specify macro expansion which is conditional upon the types of actual parameters supplied at call time. We define two constructs for doing this. The first is an  $L_E$  construct for prescribing call time expansion which is conditional upon predicates involving parameters' types. The second is an  $L_B$  construct for writing generic declarations in the macro body.

#### 3.4.1 Expansion conditional upon predicates

The  $L_E$  construct is similar to others for specifying conditional text replacement and has the form,

(3.49) if <predicate> then  $T_1$  else  $T_2$

where the predicate was described in section 3.2.2. As before,  $T_1$  and  $T_2$  each represent either a nested  $L_E$  construct denoting text or simply a bracketed segment of  $L_P$  text. The construct (3.49) expands to text  $T_1$  if the predicate is true and to  $T_2$  otherwise.

For example, assuming the replacement macro definition for the if-then-else statement in (2.37), consider the following (type 2) macro definition.

```

define $expression
  rule 'if' $expression1.bool 'then' $expression2.eval
    'else' $expression3.eval
  where match (type $expression2, type $expression3)
  means
    list
      [begin let result =],
      if match (type $expression2,int) then [0;] else
(3.50)   if match (type $expression2,bool) then [false;] else
      if match (type $expression2,str) then ["";]
      else [null;],
      $statement :
        [if $expression1 then result := $expression2
         else result := $expression3],
        [→ result end]
    end
  endef

```

Firstly, the tagging of \$expression, in the template requires that the corresponding actual call time parameter be boolean and the where clause requires that the two actual parameters matching \$expression<sub>2</sub> and \$expression<sub>3</sub> have the same type. The L<sub>E</sub> conditional in the body prescribes that the text initialising (and so declaring) the variable "result" be appropriate to the type of \$expression<sub>2</sub> (and \$expression<sub>3</sub>). For example, assuming that j and k have been declared as integers we may interpret the call,

(3.51) if j < k then j else k

as the (partially) expanded expression,

```

(3.52)  begin let result = 0;
        if j < k then result := j
        else result := k
        → result
      end

```

Note the phrase "→ result" makes the block an expression whose value is result.

For the sake of completeness we define an L<sub>E</sub> assertion form for the where clause which is analogous to the L<sub>E</sub> construct in (3.49);

the conditional assertion has general form,

(3.53) if <predicate> then  $A_1$  else  $A_2$

where  $A_1$  and  $A_2$  are nested assertions. If the predicate is true then the assertion (3.53) is satisfied if and only if assertion  $A_1$  is satisfied; otherwise the satisfaction of (3.53) depends on  $A_2$  being satisfied.

Macro expansion which is dependent on the types of actual call time parameters is best suited for the definition of generic operators. Since it is natural that the declaration of variable names local to an expanded macro body might depend on the actual call time parameters, we define a special  $L_B$  construct for writing such generic declarations.

#### 3.4.2 Generic declarations in $L_B$

A generic declaration is an  $L_B$  construct rather than an  $L_E$  construct; it has general form,

(3.54) declare <identifier> as <formal parameter ref>

Although it is an alternative form for <declaration> in the base language, it may only appear within the replacement text of a macro body. At macro expansion time the identifier is declared as a variable name having the same simple type as the actual parameter matching the formal parameter referenced in the construct. That is, (3.54) is essentially an abbreviated notation for,

```
if match (type <formal parameter>, int) then
  [let <identifier> = 0] else
if match (type <formal parameter>, bool) then
  [let <identifier> = false] else
if match (type <formal parameter>, str) then
  [let <identifier> = ""]
else [let <identifier> = null]
```

The formal parameter itself must be tagged with eval in the template so that the type of the actual parameter may be determined at call time.

Generic declarations may be found in other extensible languages; for example ALEC (Napper and Fisher, 1976) includes a form of the generic declaration. The advantage of such a facility is illustrated by the following rewriting of (3.50) for defining the conditional expression.

```
define $expression
  rule 'if' $expression1.bool 'then' $expression2.eval
    'else' $expression3.eval
  where match (type $expression2, type $expression3)
  means
    [begin
(3.56)      declare result as $expression2;
            if $expression1 then result := $expression2
            else result := $expression3
            → result
    end]
  endef
```

As another example, consider the following definition of the case statement.



```

define $statement
  rule 'case' $expression.eval 'of'
    'begin'
      stmts: {$constant.eval ':'
              others: {$constant.eval ':'} $statement';'}
      'default' ':' $statement 'end'
  where
    forall stmts:
      list
        match (type $expression, type $constant.stmts),
        forall others.stmts:
          match (type $expression, type $constant.others.stmts)
(3.57)   end
  means
    list
      [begin declare test as $expression; test := $expression;],
      forall stmts:
        list
          [if (test = $constant.stmts)],
          forall others.stmts:
            [or (test = $constant.others.stmts)],
            [then $statement.stmts else]
        end,
      [$statement end]
    end
  endef

```

Admittedly, the introduction of the generic macro does lead to a mixing of the extension language  $L_E$  with the base language  $L_B$ . Although such mixing can lead to confusion and should generally be avoided we believe that increased readability of definitions justifies this particular construct.

### 3.5 The complete macro process

If we are to define a macro evaluation process which takes account of types we must distinguish among several possible text processing modes. These modes concern the parsing of  $L_P$  text and determine the extent to which the computation tree being constructed is to be evaluated.

We say the processor is in parsing mode when no type checking is to be done. The processor operates in this mode when parsing  $L_p$  text segments in a macro body and actual parameters matching untyped formal parameters. The computation tree constructed in parsing mode is like that described in section 3.1 and contains calls to analytic routines; all analytic actions are deferred for later evaluation.

We process text in type evaluation mode when context sensitive types are to be evaluated either for type checking or expanding generic macro calls. Actual parameters restricted to specific types and type 1 macro bodies (as in section 3.3.1) are processed in this mode. Here all analytic actions in the computation tree being constructed (or copied at call time) are immediately executed. But declarations are simulated and address computation is deferred in the manner described in section 3.3.4

We say the processor is in code generation mode when compiling  $L_p$  text to pure executable code;  $L_p$  programs alone are compiled in this mode once all macro definitions have been processed. As computation trees are constructed in this mode all analytic actions and address computations are immediately performed. The pure code trees are not saved here but are output as part of the target object code deck.

These modes are hierarchical; the processor performs all functions in code generation mode which it would perform in type checking mode and performs all functions in type checking mode it would in parsing mode. Furthermore, the processor must permit the stacking (or nesting) of modes for processing nested calls. That is, it must be able to save the current mode on encountering a call which is to be processed in another mode and then return to its original mode once the call has been evaluated. We shall assume such a mechanism in the description of the process below.

The processor begins processing macro definitions in parser mode; once all definitions have been recorded, the processor shifts into code generation mode for compiling  $L_P$  programs.

The macro process for interpreting macro definitions may now be outlined as follows.

1. A macro definition is recognised against  $L_E$ 's deterministic top down grammar.
2. The target class is recorded.
3. The macro template is parsed and converted to an internal list whose structure reflects the template's structure. Portions of the list which represent sub-templates are constructed according to the syntax graphs illustrated in (2.10 - 2.12).
- (3.58)
4. If there is a where clause then that clause is scanned and the  $L_E$  assertion is parsed and translated to an abstract program tree for call time execution.
5. The macro time  $L_E$  text of the macro body is translated into an abstract program tree which will be executed at call time for controlling macro expansion. Bracketed  $L_P$  text segments are broken into tokens; those text segments which are tagged with classnames are parsed against those classnames for producing computation (sub-) trees. In the special case of type 1 macros, the entire body is parsed against the target class for producing a single tree; this is done in parsing mode unless the body is tagged with eval in which case the processor temporarily shifts into type evaluation mode.
6. If a nested macro call is encountered during a parse in step 5 then the process is temporarily interrupted, the call is evaluated and the resultant (sub-) tree is returned as a branch of the outer computation tree.

7. The target class, recorded in step 2, and the template's internal list are checked against  $L_p$ 's top down grammar to ensure that the deterministic condition is not violated.
8. Finally, the template's internal list, assertion tree, and body tree are linked together and the template's list is attached to the list of alternative BNF rules for the target class. The parser's decision table is updated to reflect the modified grammar.

Macro calls may then be processed as follows.

1. A macro call is recognised as a compound target reducing from the target class in  $L_p$ 's grammar. In general, a call is evaluated in the same mode in which it is recognised.
2. The call is parsed against the macro template's internal list structure. As sub-templates are encountered we record how they are (or are not) matched. Actual parameters are parsed against formal parameters for constructing computation (sub-) trees. Actual parameters matching untyped formal parameters are evaluated in parsing mode; those matching typed formal parameters are evaluated in type evaluation mode so long as the call itself was recognised (3.59) in either type evaluation mode or code generation mode.  
The actual parameters' computation trees, type information and information regarding the matching of sub-templates are all linked into a call time list structure which reflects the template's structure.
3. If a nested call is encountered in the evaluation of an actual parameter in step 2, then the process is temporarily interrupted, the nested call is evaluated and the resultant computation tree is returned as a branch of the tree being constructed in the outer process.

4. If the macro being called has a where clause then the abstract program tree produced for the  $L_E$  assertion at definition time is executed with the call time list of step 2 as input for determining whether the assertion has been satisfied.
5. If the assertion of step 4 is satisfied by the actual parameters of step 2 then the call is expanded. This involves executing the body's abstract program tree with the call time list of step 2 as input for producing a list of  $L_P$  tokens, computation (sub-) trees built at definition time and/or references to formal parameters. This list is parsed against the macro's target class for producing a complete computation tree; actual parameter trees are substituted for formal parameter references. The form of the final tree will depend on the mode in which the call is evaluated; if evaluated in code generation mode then a pure code tree is output as it is constructed. Note that a single computation tree for a type 1 macro body parses trivially and is merely copied across (with parameters substituted) for evaluation.

It is important to point out that all analysis, whether context free analysis (parsing) or context sensitive analysis (type checking), is done as early as possible in the macro process. The more information supplied by the user in defining his macros then the earlier this analysis may be done for producing meaningful error diagnostics.

Until now we have not discussed the scope of the macro definitions the user may define; we treat the subject of scope in the following chapter.

## CHAPTER 4

### Scope of Definition

#### 4.1 The influence of lexical scope

The scope rules for individual macro definitions will influence the nature of extensibility afforded by our system as a whole. Here we concern ourselves both with the lexical scope of macro definitions and with the lexical scope of  $L_B$  names used in such definitions.

Leavenworth's syntax macros (1966) are an integral part of his block structured base language  $L_B$ ; a macro definition may appear wherever a variable declaration may appear in a user's  $L_P$  program. The nested scope for variable names in  $L_B$  leads quite naturally to a nested scope for macro definitions; a macro call may appear in and only in that block in which the macro is defined. Although Leavenworth does not say so explicitly in his paper, we may assume macro definitions may contain variable names declared globally to such definitions. Here  $L_E$  and  $L_B$  are a single language providing end user extensibility.

An alternative to Leavenworth's scheme is to keep the extension mechanism  $L_E$  and base language  $L_B$  separate. Input consists of a file of  $L_E$  macro definitions, extending  $L_B$  to a new language  $L_P$ , followed by a user program written in the extended  $L_P$ . The scope of each macro definition in the file is the user's program and (possibly) subsequent definitions in the file. All macro definitions are independent of particular user programs; no macro definition may contain variable names not declared locally to that definition. This gives us a language definition system where a super user extends a given base language  $L_B$  for defining a fixed  $L_P$  by means of a file of  $L_E$  definitions. The end user does not retain the extension

facility in his  $L_P$  programs.

Napper and Fisher (1976) introduce a variation to the idea of definition files in the description of ALEC. In ALEC, all macro definitions must precede (and are global to) the user's program but they may contain names declared in the  $L_P$  program. Since names are not bound until call time their binding depends on the appearances of calls in the program; this two dimensional interpretation of textual scope is described in more detail in our survey (appendix 1). The essential point is the end user can define a file of macros which are peculiar to his personal  $L_P$  program. This also means some files of macro definitions may not represent 'pure' linguistic extensions but may assume the declaration of specific names in the  $L_P$  programs.

We choose to provide both an  $L_P$  definition facility utilising definition files and a nested definition facility along the lines of the Leavenworth proposal for the end user. But we wish to keep the two separate; all macro definitions which define  $L_P$  must be independent of particular  $L_P$  programs but the end user's facility will be an integral part of  $L_P$ . We describe these two facilities separately in the following two sections.

#### 4.2 The definition of $L_P$

The definition mechanisms introduced in chapters two and three together constitute our  $L_P$  definition facility. A 'program' in our system takes the form

(4.1)  $\langle L_E \text{ definition file} \rangle$   
program  
 $\langle L_P \text{ program} \rangle$

Any definitions in the (possibly empty) file are processed serially for modifying  $L_B$  for defining an extended language  $L_P$ ; if the file is empty then  $L_P$  is the unmodified base language  $L_B$ . The user's  $L_P$  program is finally compiled according to  $L_P$  as defined in the file.

The  $L_E$  file has the form

(4.2) {  $\langle \text{definition} \rangle$ ; |  $\langle \text{replacement definition} \rangle$ ; }  
      {  $\langle \text{deletion} \rangle$ ; }

where, here, the braces ({,}) indicate zero or more occurrences of the sequence they enclose. A definition has the general form

define  $\langle \text{target class} \rangle$   
(4.3)     rule  $\langle \text{macro template} \rangle$   
          [where  $\langle \text{assertion} \rangle$ ]  
          means  $\langle \text{macro body} \rangle$  endif,

a replacement definition has the form

replace  $\langle \text{target class} \rangle$   
          rule  $\langle \text{abbreviated template} \rangle$   
(4.4)     by  $\langle \text{macro template} \rangle$   
          [where  $\langle \text{assertion} \rangle$ ]  
          means  $\langle \text{macro body} \rangle$  endif,

and a deletion has the form

(4.5) delete  $\langle \text{target class} \rangle$   
          rule  $\langle \text{abbreviated template} \rangle$  endif,

where square brackets ([,]) enclose optional where clauses.



That is, a file is a sequence of zero or more definitions and replacement definitions followed by a sequence of zero or more deletions; each form is followed by a semicolon. The restriction that deletions come last in the file is a concession to implementation.

The interpretations of (4.3), (4.4) and (4.5) are as described in chapters two and three. All definitions and replacement definitions are independent of particular  $L_P$  programs; no macro body may contain names which have not been declared locally (for those reasons discussed in section 3.3.2 this cannot always be checked). Each  $L_E$  definition in the file represents a modification to a changing  $L_P$  which began as  $L_B$ ; the file taken as a whole defines the final  $L_P$  available to the end user. Since each modification to  $L_P$  involves a modification to the underlying grammar (ie an added, replaced or deleted BNF alternative) calls to macros introduced in the file will be recognised according to the modified grammar for  $L_P$ .

#### 4.3 Macro definitions in $L_P$

To this point we have kept our Algol-like base language  $L_B$  nearly separate from our definition language  $L_E$ ; the generic declaration introduced in section 3.4.2 is an exception.

We now introduce a macro definition facility for the end user. We call it a macro declaration and incorporate it in  $L_B$  as an alternative form for the syntactic class <declaration>. The macro declaration has the form

```
macro <target class>  
  rule <macro template>  
  [where <assertion>]  
  means <macro body> endmacro
```

where the square brackets indicate the where clause is optional. The target class, (optional) assertion and macro body are as defined for type 1 and type 2 macro definitions.

The interpretation of macro declarations is nearly the same as defined for type 1 and type 2 macros. The differences concern textual scope and the manner in which the grammar is modified.

The scope of a macro is that block in which it is declared, a macro declaration must appear before calls to it in the  $L_p$  program. If a macro declaration introduces a template which (in the conventional sense) conflicts with a template introduced for the target class in a surrounding block then macro calls are recognised according to the most recent declaration. No two macros introducing templates with like starting tokens for the same target class may be declared at the same block level. Thus, macro declarations have the same scope rules that names have.

The body of a macro declaration may contain names which have been declared globally. For example, assuming integer variables  $i$  and  $j$ , consider the following  $L_p$  program segment.

```

... begin
    structure new (int info, ptr next);
    let top = null;

    macro $statement
        rule 'push' $expression.int
    means
        eval : [top := construct new ($expression,top)]
    endmacro;

    macro $factor
        rule 'pop'
    means
        eval : [begin
(4.7)             let result = 0;
                    if top ≠ null then
                        begin
                            result := info (top);
                            top := next (top)
                        end
                        → result
                    end]
    endmacro;

    push i+1;
    push j+1;
    push pop * pop;
    write pop

end ...

```

Since we tagged the bodies of both (type 1) macro declarations with eval they can be diagnosed for type violations in the context of declared names "new" and "top".

Example (4.7) illustrates the utility of the  $L_p$  macro declaration. With it the end user may define local macros which do not represent pure linguistic extensions to  $L_B$  but are peculiar to his own problem. The push and pop constructs might be implemented with procedures in a conventional programming language.

Although the macro declaration is a construct of  $L_B$  we have chosen to disallow the nesting of macro declarations within the bodies of other macro declarations and macro definitions. This is not a concession to implementation; in fact, our implementation would be relatively easy to modify should we have wished to include nested declarations. Rather, this restriction is made so as to simplify the language itself; we discuss this point further in Chapter 6.

The macro declaration need not appear in the extended language  $L_P$ . A super user may deny the facility to the end user, and so fix  $L_P$ , by including the following deletion to his  $L_E$  definition file.

(7.8) delete \$declaration  
      rule 'macro' ... endef

If he does decide to carry the macro declaration along to  $L_P$  then all such declarations take effect as soon as they are recognised in the  $L_P$  program - even when they are written as actual parameters of macros redefining surrounding  $L_B$  forms, eg <program> and <sequence>. This permits subsequent macro declarations to make use of the newly defined  $L_P$  forms in the definitions of their macro bodies.

To summarise, we may think of the form illustrated in (4.1) as the topmost construct in our extension language  $L_E$ . That is, an  $L_E$  program consists of a (possibly empty) file of  $L_E$  definitions, replacements and deletions, followed by the token program, and, finally, the  $L_P$  program. The language  $L_P$  is  $L_B$  modified as specified in the  $L_E$  definition file. The output produced for our  $L_E$  program is a file of object code compiled for the  $L_P$  program.

## CHAPTER 5

### Implementation

#### 5.1 Introduction

Our task in implementing the system we have described is to write a compiler-processor for interpreting  $L_E$  programs conforming to (4.1). This in turn involves writing a compiler for the base language  $L_B$  and implementing the macro process outlined in section 3.5 together with macro replacements, macro deletions, macro declarations and the special generic declaration. In the sections which follow we first discuss the representations of the underlying grammar and the syntax analysis method for the system as a whole and then outline the implementation of the system's various components.

Firstly, we chose Algol W (Wirth and Hoare, 1966) as the implementation language since it is of a high enough level for representing the data structures we required and because it is fully supported at St. Andrews. We might have used one of the systems programming languages available to us, eg BCPL (Richards, 1969), PL360 (Wirth, 1968) or the UNIX systems language C (Ritchie, 1973); yet these have neither the string and list facilities nor the diagnostic aids contained in Algol W. We would like to have used Ron Morrison's Algol R (1978) since it contains all of the data structures we require and because its compiler produces fairly compact code; but we were uncertain as to when its implementation would be completed when we started out. In the end we opted for the more readily available and more tested Algol W.

## 5.2 Representing the underlying syntax

Because our system is based on the syntactic macro, syntax analysis dominates its implementation. We discuss in this section the problems of lexical analysis, the representations of grammars for  $L_E$  and  $L_B$  and our overall parsing method. We defer discussion on grammatical modification to  $L_B$  until the relevant sections covering the macro process.

### 5.2.1 Lexical analysis - the micro syntax

The  $L_E$  processor and  $L_B$  (and  $L_P$ ) compiler both share the same lexical analyser. This is a procedure (called nextsym) which fetches the next incoming symbol; this symbol is taken either from the source text (calling sourcesym in inputmode) or from a symbol list of an expanded macro body (when not in inputmode). Incoming symbols include basic tokens (uppercase words and punctuation characters), quoted tokens (for templates), classnames, identifiers and constants.

The interpretation of these symbols depends on the process ( $L_E$  processor or  $L_B$  compiler) calling for them. For example, the  $L_E$  processor may treat integer, string and boolean constants differently than would the  $L_B$  compiler; for  $L_B$ , integers, strings, booleans and the null pointer (not used by  $L_E$ ) are constant values which must be compiled as part of the object code.  $L_E$  uses classnames in its definitions (eg as formal parameters and target classes) while  $L_B$  interprets classnames as formal parameter references encountered in the parsing of  $L_P$  text segments in a macro body. These interpretations are determined in the  $L_E$  processor or  $L_B$  compiler rather than in lexical analysis.

Of course, as much semantic information as possible is determined in lexical analysis, eg the string of characters representing an identifier or the integral value of an integer. We also had to define the interface between the  $L_B$  syntactic classes, \$identifier and \$constant,

and the lexical identifiers and constants in such a way that the syntactic classes might be redefined by macros; we discuss this below in section 5.3 dealing with  $L_B$ .

### 5.2.2 Parsing $L_E$

Our definition language  $L_E$  is parsed (and interpreted) by the one pass method of recursive descent which has been popularised by various implementations of Wirth's Pascal (1971), eg Ammann (1973), and very clearly described by Griffiths and Peltier (1968). That is, the grammar (and semantics) for  $L_E$  is embedded in a series of procedures which, when called, recognise and interpret corresponding  $L_E$  constructs.

For example, the procedure for interpreting input to our system, defined at (4.1), is written in Algol W as follows.

```
procedure main;  
begin  
    foreword;  
    tabulate_lb  
(5.1)  definitions;  
    mustbe("PROGRAM");  
    compile(program);  
    postscript  
end
```

Here, procedure definitions is called for recognising and interpreting the  $L_E$  definition file; procedures foreword, tabulate\_lb and postscript perform administrative tasks and procedure mustbe scans its parameter as a lexical token in the source text. Procedure compile (described below) is called for compiling object code for the  $L_P$  program.

Because  $L_E$  is relatively separate from  $L_B$  and  $L_E$  is unmodifiable, the use of recursive descent is straightforward. There are procedures for dealing with all  $L_E$  entities.  $L_E$ 's grammar is given in Appendix 2 and its interpretation is covered in sections below.

### 5.2.3 Parsing $L_B$

Like  $L_E$ ,  $L_B$  is mainly parsed (and interpreted) by the method of recursive descent; we define a separate procedure for parsing (and producing code for) each  $L_B$  syntactic form. But, because we shall want to modify  $L_B$ , alternative forms (sub-targets) to a target syntactic class are recognised by way of an amendable parsing decision table.

The construction of our parsing table is based on that proposed by Aho and Ullman (1972) for tabular parsing of LL(1) grammars. But, since no  $L_B$  form nor any macro template may be matched by the empty string, our table need only take account of a form's possible starters (possible initial identifying tokens) and can ignore possible followers. For this reason the table is easily constructed for  $L_B$  and easily modified for reflecting  $L_P$ .

The table takes the form of a two dimensional matrix of pointers with twelve rows, each corresponding to a target class of  $L_B$ , and enough columns (this is variable) to correspond to each possible starter of an  $L_B$  form or macro template. To each pair consisting of a target syntactic class and a possible starter for an alternative form defined for that class there is a pointer to a record describing the  $L_B$  form (or macro).

For example, to the  $L_B$  form,

(5.2) `<statement> ::= if <expression> then <statement>`

there corresponds an entry in the table indexed by the row corresponding to target class `<statement>` and column corresponding to the starter if. This entry is a pointer to a record defined in Algol W as

(5.3) `record lb (integer ruleno;  
                  logical lbdel)`

The first field of this record is an index (through a case statement) of the (recursive descent) procedure for recognising and interpreting



the  $L_B$  form in (5.2). The second field is used for  $L_E$  deletions, described later.

Parsing decisions are made via the procedure compile. For example, to compile source text against the target class <statement> we would invoke the call,

(5.4) compile (statement)

If the token if were the next incoming symbol then compile would (via a decision procedure) invoke the procedure referenced by the entry corresponding to (5.2). Ignoring semantic routine calls, the Algol W procedure for (5.2) is as follows.

```
procedure lb_if_statement;  
begin  
    mustbe("IF");  
(5.5)    compile(expression);  
    mustbe("THEN");  
    compile(statement)  
end
```

This procedure would also have to include calls to semantic procedures but, more importantly, although the expression and statement are recursively compiled, the parsing decisions must be made with reference to the decision table via the procedure compile.

Since macro time parameters (and body text parsed at definition time) may be encountered as input to an  $L_P$  parse, the table must allow for syntactic classes as starters as well as for token starters.

Initially, the table is constructed for  $L_B$  by a procedure (called tabulate\_lb) based on that of Aho and Ullman and using Warshall's algorithm (1962) for computing the closure for the target-starter relations. All entries corresponding to  $L_B$  forms point to records of the form in (5.3); all other entries are initialised to null. If a null entry is encountered by procedure compile then a syntax error is reported. Once modifications are introduced with macro definitions, the modified table will contain entries to an alternative record for describing the corresponding macro

definition; we discuss these modifications later.

Some parsing decisions which are local to specific  $L_B$  forms, eg recognising a list of terms and operators in an expression, are, like  $L_E$  parsing decisions, made within the procedures. For all such parsing decisions we employ a syntax error diagnosis and recovery algorithm proposed by D A Turner (1977) and based on the scanning procedure mustbe. Here an initial scanning error is reported and then temporarily ignored; a second scanning error either forces a match or scans to some relevant delimiter, eg ';', end or endef . Although recovery is not perfect, the method is general enough for use with a grammar like  $L_B$ 's which is continually being modified.

### 5.3 The compiler for $L_B$

We said in the last section that  $L_B$  text is compiled by a method of recursive descent which makes use of a parsing table for deciding the alternative  $L_B$  forms for target classes in a top down parse. In this section we cover those semantic actions of the  $L_B$  compilation process which pertain to the macro process.

Firstly, we have kept the language  $L_B$  relatively separate from  $L_E$ , if we leave macro declarations and generic declarations aside then  $L_B$  could pretty well stand on its own. The lexical analyser, parsing table, recursive recognition routines and the semantic routines we have defined form the basis of an  $L_B$  compiler. We have written the compiler-processor in this way not only for the sake of modularity but to demonstrate that  $L_B$  is not the only language on which our facility might be based. Our method might be based on any language with an LL(1) syntax. Since the macro declaration and generic declaration are  $L_B$  constructs they might or might not be fitted into an alternative base language depending on its scoping and typing qualities. True, the  $L_E$  constructs for identifying simple and structured types would require modification but (as we shall see below) our data structures

which represent the types of names internally are not defined so as to be peculiar to  $L_B$ . We chose  $L_B$  not as a general purpose programming language but for illustrating some of the definitional and diagnostic facilities possible in a syntactic macro facility.

The  $L_B$  (and so  $L_P$ ) compiler produces code for an abstract stack machine (with a heap) described in Appendix 3. This code can either be interpreted directly or compiled to some hard machine code. For illustrative purposes we list the code generated, line by line, with the source listings produced for  $L_P$  programs.

Our major interest here is the interpretation of the analytic (type checking) and constructive (code generation) semantic actions between which we distinguished in section 3.1. Consider the following Algol W procedure for interpreting the  $L_B$  if statement (abbreviated in (5.5)).

```

      procedure lb_if_statement;
      begin
        mustbe("IF");
        compile(expression);
        popcheck(bool);
(5.6)  stack_newlab;
        emit("JUMPF ");
        emit_toplab;
        mustbe("THEN");
        compile(statement);
        set_popped_lab
      end
```

We have already discussed the scanning procedure, mustbe, and the recursive procedure, compile, in the last section; the other procedure calls represent semantic actions. Procedure emit invokes the only purely constructive action; its effect is to generate the object code instruction mnemonic given as its parameter. Procedure popcheck invokes a purely analytic action. Its effect is to check the type given as its parameter for equality with the type last stacked atop a compile

time type stack for a previously compiled value; its side effect is to pop that type from the stack. Procedures `stack_newlab`, `emit_toplab` and `set_popped_lab` are examples of semantic actions with both analytic and constructive functions which cannot be distinguished at this level; these particular procedures calculate and emit object code branch addresses.

If we assume a stand alone compiler for  $L_B$  and ignore  $L_E$  then we may execute the if statement's procedure (5.6) as follows.

1. Scan the if in the source text;
2. Compile and generate object code for the expression which follows (leaving its type atop the type stack);
3. pop the type stacked for the expression in step 2 and check that it is type bool;
4. create a new unique label for use in the object deck and put (5.7) it on the label stack;
5. generate the object code instruction, JUMPF;
6. generate the label, created in step 4, as an argument to JUMPF;
7. scan the then; compile the statement; and
8. pop the label stacked in step 4 and generate it as a label in the object deck.

If we execute (5.6) with this interpretation then there is little need to distinguish between analytic actions and constructive actions. The abstract stack machine code generated is simply,

```
[code for expression]
(5.8) JUMPF L
      [code for statement]
L:
```

where label L is unique in the object deck.

This interpretation is precisely that which our compiler-processor uses for executing (5.6) in code generation mode. But when in either parsing mode or type evaluation mode, where it is constructing a computation tree for an actual parameter to a macro call or a text segment in a macro body, the execution of some steps in (5.7) must be deferred.

In parsing mode, procedure calls which invoke analytic actions, eg popcheck, and calls which invoke address computations, eg stack\_newlab, are linked to a list representing the relevant computation tree; links for these calls are represented by specially formatted records. The call to procedure emit, invoking a purely constructive action, causes its instruction parameter to be linked to this same list. When the computation tree is copied for macro expansion, if the processor is in the appropriate mode then the calls and instructions which form the links are executed and output; otherwise they are copied to the new computation tree which is being constructed.

In type evaluation mode, procedure calls invoking analytic actions are executed immediately for checking types atop the compile time type stack. Calls which invoke address computations and the instruction parameters to emit are linked to the computation tree. Calls which invoke both constructive and analytic actions (where the constructive actions are not constants of the syntax) must be treated as analytic actions.

During compilation (of a program and/or a macro body in type evaluation mode) we maintain a nest of environments using a linked list. Each environment in the nest corresponds to a nested block in the  $L_P$  text and contains a list of names declared in the environment together with their types (and possibly their stack or object code addresses). The environment list is queried by the  $L_B$  analytic procedures for stacking and checking types on the compile time type stack. The type stack and the types themselves are represented by linked records with

a format declared in Algol W as follows.

(5.9) record type (reference (type) hdt, tlt)

Fields hdt and tlt each point either to another type or to null.

The type stack is represented as a list of these records linked together by their tlt fields. The record structure (5.9) is used for representing all simple and structured  $L_B$  types.

Initially, the system creates records named int, bool, str and ptr for simple types, records named constructor and field for representing structured types and records named any, any\_constructor and any\_field which are used by the processor for recovering from type errors. Subsequently, types int, bool, str and ptr are each represented by a pointer to the appropriately named record. Structured types are almost as easily represented. For example, the following  $L_B$  declaration introduces four names into its environment.

(5.10) structure treenode (int info; ptr left; ptr right)

The internal representations for the types of these names are shown in (5.11); each name used in a field is a pointer to the single record initially created with that name.

```

      treenode : (constructor, ↪ (int, ↪ (ptr, ↪ (ptr, null))
(5.11)      info : (field, int)
            left : (field, ptr)
            right : (field, ptr)

```

The validity of field names applied to the pointers (to structures) in  $L_B$  is checked at run time.

$L_E$  constructs which denote types are translated to internal representations with structures identical to those given above. With this internal structure, types are compared for equality by recursively defined type checking procedures.

We claim no originality for our representation; it is a derivative of that used by Turner and Morrison in the implementation of their

language, Algol S (1976). We describe it only to demonstrate the ease with which other kinds of types might be represented internally should one wish to apply our facility to an alternative base language. For example, procedure types might easily be incorporated if based on structures constructed from two additional records named procedure and parameter.

A final semantic action pertaining to the processor concerns the treatment of constants and identifiers recognised by the lexical analysis procedure, nextsym. For the sake of completeness, the syntactic classes <constant> and <identifier> must be redefinable (as target classes) and must be permitted as formal parameters in the macro templates for  $L_E$  definitions. Therefore, the  $L_B$  compiler must parse (and interpret) them, like any syntactic class, via the procedure compile. Initially, <constant> and <identifier> each reduce to a single  $L_B$  form as defined by the procedures lb\_constant and lb\_identifier respectively.

The procedure for  $L_B$  constant forms is as follows.

```
procedure lb_constant
  begin
(5.12)   loadc(the_constant, its_type);
          stacktype(its_type);
          nextsym
  end
```

Values for the\_constant and its\_type are determined by the lexical analyser. Procedure loadc generates code for loading the constant, stacktype is an analytic procedure for stacking its type on the type stack and nextsym fetches the next input symbol. The syntactic class <constant> may be redefined by a macro so long as the body expands to the  $L_B$  constant form.

```
procedure lb_identifier;
  begin
(5.13)   stackid(nom(the_id,,,null));
          nextsym
  end
```



Again, the value for the id is determined by lexical analysis.

Procedure `stackid` is an analytic action for putting an identifier atop the identifier stack; the extra fields in the record used as a link in the list representing the stack is a result of our having to share single record formats among tasks; Algol W allows only fifteen formats. Once on the stack, the identifier is accessible either to a surrounding  $L_B$  form, eg a declaration form, or to  $L_E$  as an actual parameter or macro body. Like `<constant>`, `<identifier>` may be redefined by a macro so long as its body expands to the  $L_B$  identifier form.

Having covered the pertinent semantic actions executed by the underlying  $L_B$  compiler we may now discuss our implementation of the macro process.

#### 5.4 The macro processor

Many  $L_E$  definition forms, eg sub-templates and control constructs in the macro body and the where clause, are recursively defined; this makes our interpretation of these forms by recursive descent particularly appropriate. In section 5.4.1 we discuss the construction of internal structures for, and the implementation of, the macro definition process outlined in (3.58). In section 5.4.2 we discuss the implementation of the calling process, outlined in (3.59), with reference to the structures constructed in 5.4.1.

##### 5.4.1 Interpreting the macro definition

###### 5.4.1.1 The macro template

Once we have recognised a macro definition (step 1 of (3.58)) in the interpretation of  $L_E$  program and have recorded the target class (step 2), we (recursively) scan and construct an internal list structure for the macro template (step 3). The internal list structure is constructed from links of three record formats for representing basic tokens, formal parameters and sub-templates. Basic tokens are simply represented by a record format (sym) containing a string field for



the token and a link to the next record.

For representing formal parameters we use a record whose format is declared in Algol W as follows.

```
(5.14)  record class (integer clrep;  
          string cname;  
          logical eval;  
          reference (type) cltype;  
          reference (sym, class, subt) clnext)
```

The field name clrep contains the index for the relevant syntactic class in the parsing table (used by procedure compile) and cname contains the parameter's name. Eval is true only if matching actual parameters are to be type checked and cltype points to the type (if any) with which the parameter is tagged. Finally, clnext points to the next token, parameter, sub-template or null.

For representing sub-templates we use a record with the following Algol W format.

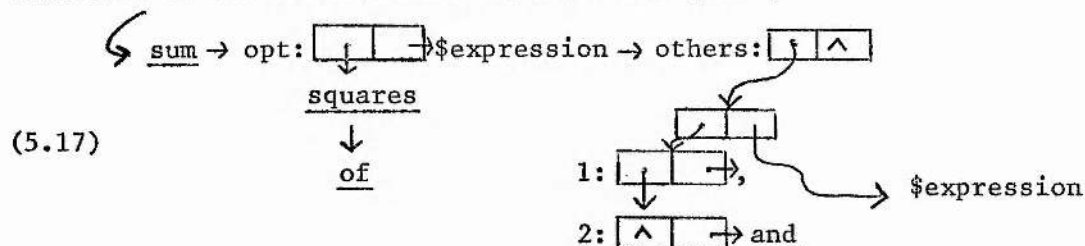
```
(5.15)  record subt (integer stype;  
          string sname;  
          logical inscope;  
          reference (sym, integral) first;  
          reference (sym, class, subt) this, that)
```

Here, stype indicates the form of sub-template, sname contains its name and inscope is used for processing the body (described below). First points to a list of possible starters of the (recursively interpreted) nested template(s). The starters (tokens and syntactic classes) are copied from the nested template(s) but no closure is computed at this point; these are used for making local parse decisions at call time. For optional and list sub-templates, 'this' points to the nested template. For an alternative sub-template, 'this' points to a list of records of the form (5.15); each of these in turn points to the alternative nested template. Finally, 'that' points to the next token, parameter, sub-template or null.

As an example, consider the following template which was used in (2.18) for introducing a new expression form.

(5.16) 'sum' opt: (? 'squares' 'of' ?) \$expression  
 others: (\* ('|''and') \$expression \*)

If we ignore the details, the following sketch shows the overall structure of the internal list created for (5.16).



#### 5.4.1.2 The assertion

If a where clause is recognised following the template then we translate the  $L_E$  assertion into a macro time program tree (step 4 of (3.58)). At macro call time this tree will be traversed (as directed by the sub-templates matched by the call) for verifying the predicates (at its leaves) in order to verify the assertion as a whole. We use one record format (mctree) for the control tree and another format (pred) for representing compound predicates. Because the control tree and parameter references for assertions are identical to those used in the macro body (with predicates replacing  $L_p$  text segments), we do not discuss them here. Predicates are simply represented internally with tree structures reflecting their syntax.

#### 5.4.1.3 The macro body

The interpretation of the macro body (steps 5 and 6 of (3.58)) is straightforward for type 1 macros. In this case we compile the body against the target class for producing a computation tree. Compilation is done in type evaluation mode if the body is tagged with eval and in parsing mode otherwise. Parameter references must be

translated to actual parameter references (used at call time) which are linked into the computation tree; we discuss parameters below.

Firstly, the procedure for producing a computation tree is as follows; since this procedure may be invoked recursively, eg in the expansion of nested calls, some global information must be saved.

1. We stack both the current processor mode and the global current code tail (a pointer to the tail of a computation tree list being constructed; see step 3).
2. We set the current processor mode according to our wishes, either to parsing mode or to type evaluation mode, and we set the current code tail to point to an initial link, call it code, which is local to this procedure.

(5.18)

3. We then call the procedure compile, described in section 5.3, with our target class as its parameter. This will compile subsequent source text against our target class; all code (and deferred analytic routine calls) which is produced via compile will be linked to the tail of our computation tree by way of the current code tail. Further, if we are compiling against a target class which represents a value in type evaluation mode then a resultant type will be left atop the compile time type stack.
4. We restore the current processor mode and current code tail to their former values from the stack.
5. Code now points to the computation tree we require. If a type was left on the type stack in step 3 then we must keep it with the computation tree as it may be required by a surrounding process.

The procedure compile is the interface between our macro processor and  $L_B$ . When not in code generation mode, all code produced by  $L_B$  procedures (or  $L_P$  macro calls) is copied to the end of our computation

tree by way of the current code tail.

We translate the body of a type 2 macro into a macro time program tree. The tree's structure reflects the structuring of  $L_E$  control constructs (eg given, forall, list, choosing and if) in the macro body. The leaves of the tree are  $L_P$  text segments. At macro call time, the tree is traversed (as directed by which syntaxes are matched in the template) for choosing the  $L_P$  text segments which are to be expanded as part of the replacement tree.

The macro time control tree is constructed from links with the Algol W record format,

```
record mctree (integer mctype;  
(5.19)          reference (parm, pred, integral) info;  
                reference (mctree, pred, integral, list) lbr, rbr)
```

Each link with this format represents one of the  $L_E$  control constructs. The field mctype is used to denote the particular  $L_E$  construct being represented. Info contains a reference to the information upon which the particular text replacement decision is to be based. For example, in the record link representing the  $L_E$  construct,

```
(5.20)  given opt then <text1> else <text2>
```

info contains a reference to a node for the optional sub-template opt in an actual parameter list produced at call time. This node will be queried at expansion time to see whether or not opt was matched by the call. Fields lbr and rbr represent branches the processor follows for expanding text based on the information gathered through info; eg in (5.20) for expanding <text<sub>1</sub>> if opt is matched and expanding <text<sub>2</sub>> if it is not matched. The branches may be either computation trees, produced for syntactically tagged text segments using procedure (5.17); lists of tokenised text, produced for untagged text; or nested record links which represent nest  $L_E$  constructs (predicates may be used here for assertions).

We also , at definition time, construct references to (future) call time actual parameter lists for all formal parameter references in the  $L_P$  text (and predicates) and all references to sub-templates which are used in the  $L_E$  control constructs. All such references are checked against the template for ensuring validity. The logical field, inscope, of the record (subt) representing sub-templates is used to flag impossible references. For example, the inscope field for opt in (5.20) is set to true before  $\langle \text{text}_1 \rangle$  is processed to signify the templates nested in opt is in scope; upon leaving  $\langle \text{text}_1 \rangle$ , the inscope field is reset to false. Therefore, any references to parameters or sub-templates nested within opt will be flagged as being out of scope. A similar method is used for checking the scope of references into alternative sub-templates and list sub-templates. Since all references are validated at definition time, references need not be checked at call time.

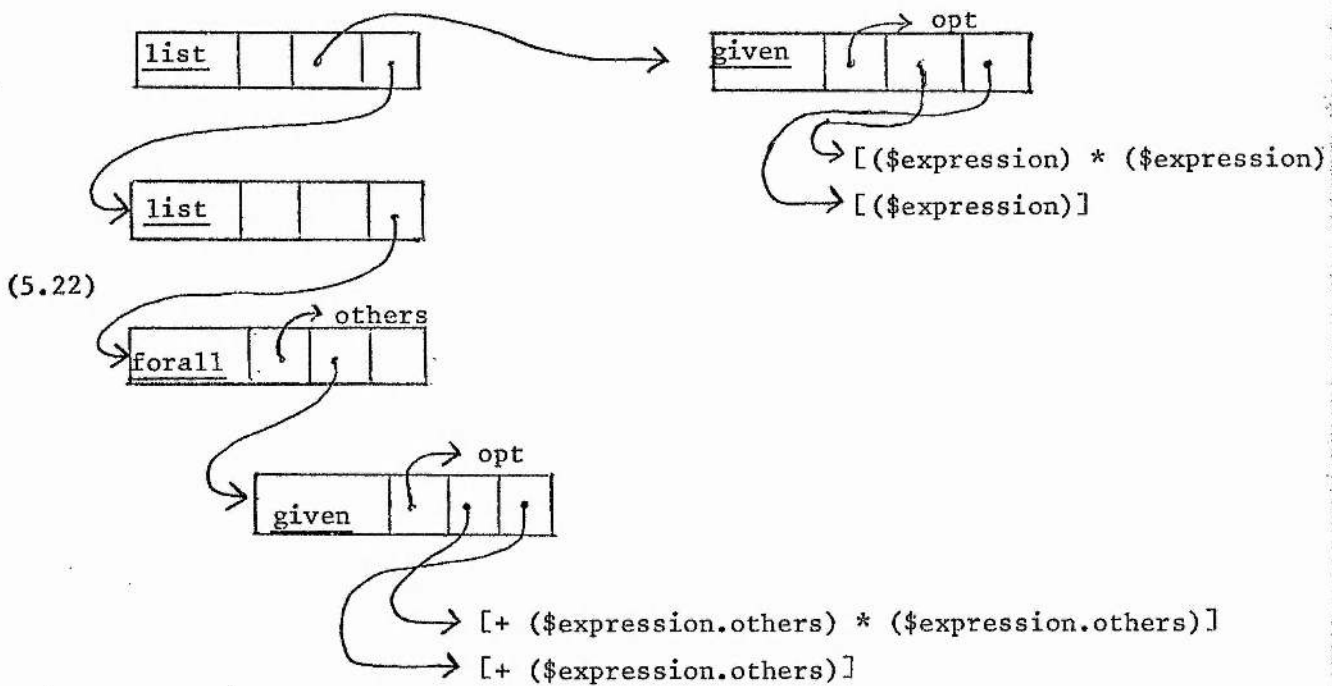
For a look at the structure of a macro time program tree for a type 2 body let us consider the macro body corresponding to the template (5.16).

```

(5.21)  list
        given opt
          then [($expression) * ($expression)]
          else [($expression)],
        forall others:
          given opt.others
            then [+(($expression.others) * ($expression.others)]
            else [+(($expression.others)]
        end

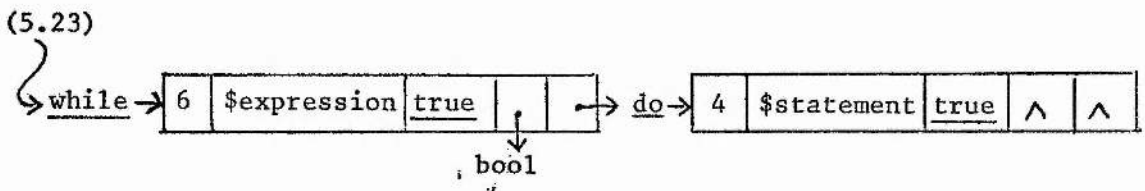
```

The macro time program tree corresponding to (5.21) is illustrated below.

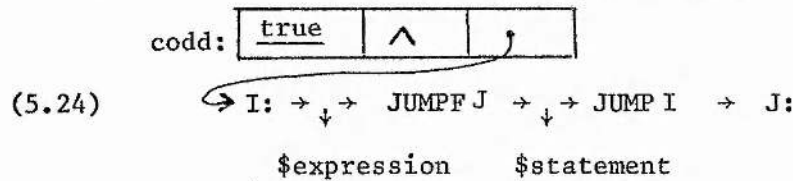


Here, the  $L_p$  text segments must be represented as symbol strings and must be parsed at call time since they are not syntactically tagged in (5.20). If we had tagged the first two segments in (5.21) with  $\$term$  then they could have been replaced with their computation trees in (5.22).

Finally, we illustrate the internal structures constructed for both the template and body of the type 1 macro definition (3.40) for introducing the while statement. Here, we assume the body has been tagged with eval so that all types may be checked at definition time. Referring to the record format (5.14) for parameters, the template is represented as,



and the macro body is represented by the following simple computation tree (the record (codd)) is a descriptor for code trees.



Note that I and J represent actions for computing addresses, eg stack\_newlab.

#### 5.4.1.4 Modifying the underlying grammar

Once we have recognised a macro definition and constructed the relevant structures we must check that it does not introduce a conflict in the parsing table (step 7 of (3.58)) and modify that table (step 8) to reflect the introduction of the new  $L_p$  form. These two tasks are performed concurrently.

Firstly, we create a descriptor for our new macro with the following Algol W format.

(5.25)      record macro (integer lev;  
                                  logical del; err;  
                                  reference (sym, class, subt) templ;  
                                  reference (mctree, pred, integral) assrt;  
                                  reference (mctree, codd) body;  
                                  reference (macro, lb) previous)

Fields templ, assrt and body point to the structures we created for the template, assertion and macro body. -Err is set to true only if an error was encountered in creating these structures in order to prevent expansion at call time. Fields lev, del and previous are used for deletions, replacements and macro declarations.

We compute all of the possible starters for the macro template by examining the template itself and referring to the parsing table (when the template starts with a syntactic class). Column entries are created for newly introduced starting tokens. At this point, we



check the target class against the template's starters for detecting immediate left recursion.

We then go through the column in the parsing table for the target class and for each row corresponding to a starter we point that entry to our macro descriptor. We do the same thing for all columns corresponding to syntactic classes for which our target class is a possible starter. If, during this process, we attempt to point a non-null table entry to our descriptor then we have a conflict; the error is reported and we leave the entry unchanged.

Once this process is completed macro calls may be recognised in the context of a normal  $L_P$  parse by way of procedure compile; in this case the new  $L_P$  form is described by a descriptor of the format (5.25) just as  $L_B$  forms are described by descriptors of the format (5.3). We must emphasise that we may use the above modification process only because no  $L_P$  form may be matched by the empty string; this means that followers, required for conventional LL(1) parsing schemes, need not be taken into account.

#### 5.4.2 Interpreting macro calls

##### 5.4.2.1 Recognition

The compiler recognises the start of a macro call (step 1 of (3.59)) as an alternative  $L_P$  form for some particular target class by way of procedure compile and the parsing table. The call itself is parsed against the internal list structure which we constructed for the macro template (steps 2 and 3) at definition time. Basic tokens in the template are scanned using procedure mustbe; this permits macro calls to take advantage of the syntax error and diagnostic facilities afforded to  $L_B$  by mustbe.

Actual parameters are compiled against the syntactic classes



of corresponding formal parameters in the template for producing computation trees using the procedure outlined in (5.18). The `clrep` field in the formal parameter's record is used as the argument to procedure `compile`. This compilation is done in type evaluation mode if the formal parameter's `eval` field is true and in parsing mode otherwise. If the `cltype` field points to a type then that type is checked against the type computed atop the compile time stack. In any case, a type computed on the type stack is removed and held together with the computation tree representing the actual parameter in a descriptor (`codd`) like that used in (5.24). At this level, there is no distinction between macro calls and  $L_B$  forms nested in the actual parameter.

Parsing decisions made in a sub-template are based on the starters in field `first` of the sub-template's record. As stated in section 2.2.2.1, the nested template of optional and list sub-templates and the first alternative template in an alternative sub-template are always matched where possible. In terms of the field names of the record (5.15), the parser generally tries 'this' way before 'that' way.

An actual parameter list, reflecting the relevant structure of the macro call, is constructed as the call is recognised. A standard parameter descriptor is created for each formal parameter and for each named sub-template which is encountered in the macro template (or a nested template) during recognition. For formal parameters, this descriptor contains such information as the formal parameter's name, its syntactic class, a pointer to the code tree descriptor (`codd` - built for the corresponding actual parameters) and a pointer to the next parameter descriptor. For sub-templates, this descriptor contains the sub-template's name, information concerning if and what nested

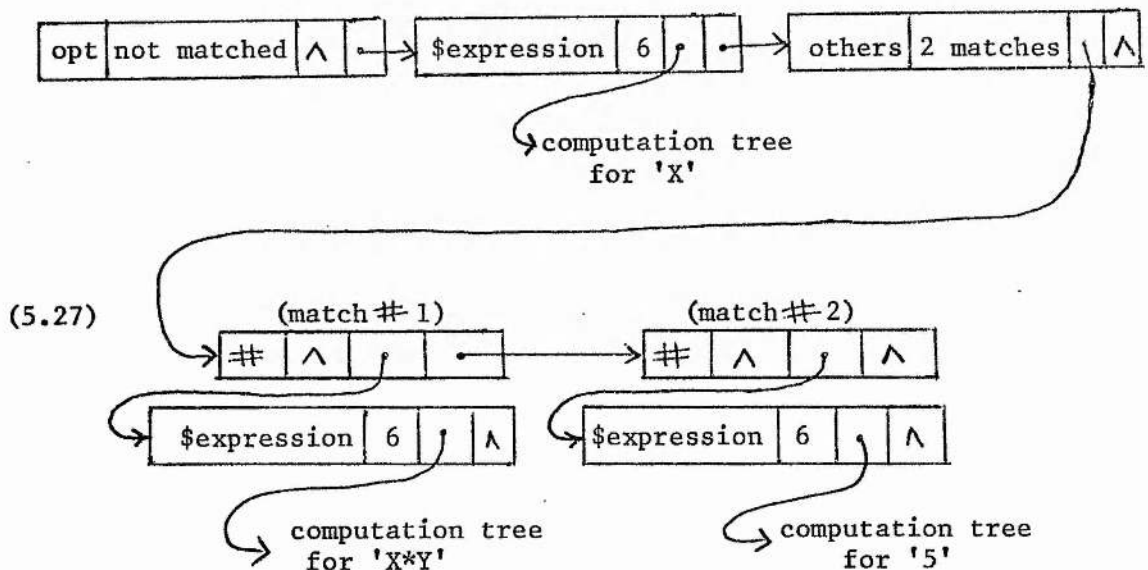
template was matched, a pointer to the actual parameter list corresponding to the nested template and a pointer to the next parameter descriptor.

The sort of information kept (in the first descriptor field) for a sub-template depends on what type it is. For an optional sub-template we would record whether or not the option was matched. For an alternative sub-template we would record the alternative chosen. For a list sub-template we would record how many times the nested template was matched (and produce an actual parameter list for each match).

For example, consider the following call to the sum macro whose template is given in (5.16); since all formal parameters are untyped, all actual parameters are compiled in parsing mode.

(5.26) sum X, X\*Y and 5

The actual parameter list constructed for (5.26), detailing only the parameter descriptors, is illustrated below.

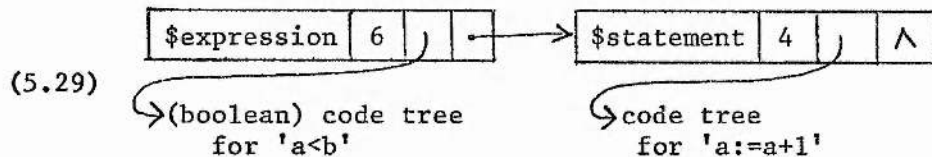


Note that the unnamed alternative sub-template in (5.16) is not reflected in this structure; this is because no (macro time or compile time) semantic information is contained in it.

Consider the following call to the type 1 macro defined in (3.40); here all parameters are tagged for type checking so actual parameters may be compiled in type evaluation mode.

(5.28) while a<b do a:= a+1

The actual parameter list constructed for 5.28 is simply,



Again, only the important information is kept in the parameter list.

This method by which we internally represent macro calls is partly based on Ollongren's language transformations (1974) where irrelevant (structuring) symbols, eg basic tokens and the alternative sub-template in (5.16), are removed.

#### 5.4.2.2 Assertion verification

If the macro has an assertion (recognised in a where clause at definition time) then the macro time program tree which is constructed for it must be evaluated for ensuring that the assertion holds (step 4 of (3.59)). For doing this we apply a recursive function for traversing the tree, as directed by decisions made at the (mctree) nodes, and evaluating the predicates at its leaves. The traversal decisions are based on information supplied in the actual parameter list. Error messages are emitted where directed by nodes for the assert construct; if an error is detected for a macro call which is nested in another expanding macro then a warning to that effect is emitted with the error message. The predicate (sub-) trees are also recursively evaluated. Boolean values returned for the predicates are passed up

the program tree to its root so as to compute the truth or falsity of the whole assertion. A macro call is expanded if and only if its assertion is true; null assertions are always true.

#### 5.4.2.3 Macro expansion

A macro call is expanded (steps 5 and 6 of (3.59)) in that processor mode in which it was recognised.

For type 1 macros, expansion is substantially a copying process. The (linear) computation tree constructed for the body at definition time is copied, beginning at its first element, for output if in code generation mode and to the (outer) computation tree being constructed otherwise. As the processor encounters links representing actual parameter references, the computation trees constructed for corresponding actual parameters are copied in their place. If the call being expanded is nested in another macro's definition, actual parameter trees may contain nested parameter references; nested parameter references are simply copied across to the outer tree. In type evaluation mode, analytic routine calls are not copied but are executed for checking types; in code generation mode, all analytic and address calculating routines are executed. Error messages generated by these routines are amended in the source listing for indicating their nesting in the call. Finally, if in type evaluation mode, any type associated with the body's computation tree is left atop the compile time stack. This copying process is performed by a single procedure. Therefore, the code produced for the type 1 macro call in (5.28) is simply

```
      I: [code for 'a<b']
          JUMPF J
(5.30)      [code for 'a := a+1!']
          JUMP I
      J;
```

For a type 2 macro, expansion requires an evaluation of the macro time program tree constructed at definition time. Recall, the leaves of this tree are either lists of tokenised  $L_p$  text or computation trees, both of which may contain actual parameter references. We first flatten this program tree for producing a linear source list and then compile this source list.

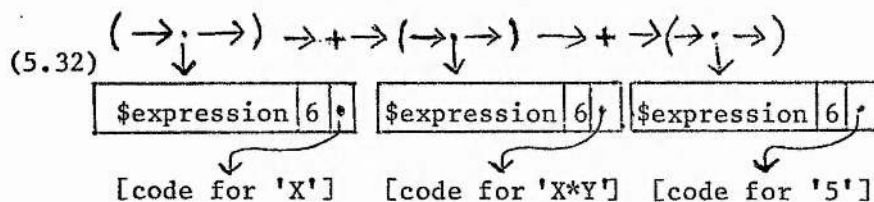
Our process requires a global boolean, called input, and a global pointer, called source. Input determines from where the lexical procedure, nextsym, gets its next input symbol; input is true if that symbol is to be scanned from the program source text and false if it is to be peeled off the list pointed to by source. Source points to the linear source list we produce from the flattened program tree. We may then expand our type 2 macro by the following procedure.

1. The current values for input (initially true) and source are saved on a stack for protecting any surrounding expansion process.
  2. For flattening the program tree we recursively traverse it as dictated by decisions made at the (mctree) nodes and based on information obtained from the call's actual parameter list. Links found at the leaves for tokens, computation trees and actual parameter references are copied to the end of our new source list. Links for actual parameter references are modified to point directly to the computation trees of corresponding actual parameters. The result of this process is a list of tokens and computation trees of specific syntactic classes.
  3. We point source to the source list produced in step 2 and set input to false for redirecting the lexical procedure nextsym.
  4. We may now apply procedure compile to the macro's target class for compiling the source list in the processing mode in which the call was recognised. Once taking their syntactic classes into account for parsing purposes, compile copies the computation
- (5.31)

trees by the procedure we described for expanding type 1 macros. Any syntax error messages emitted during compilation are amended to indicate their nesting in the call.

5. Finally, the original values for source and input are restored from the top of the stack for reflecting the surrounding lexical environment.

For example, the source list produced in step 4 of (5.31) for the macro time program tree (5.22) and actual parameter list (5.27) is illustrated below; since opt was not matched, the list is relatively simple.



The source list in (5.32) is compiled against the target class, <expression>, taking into account the syntaxes of the computation trees, for generating code (in code generation mode) or for creating a computation tree (otherwise).

This concludes our discussion of the implementation of the macro process outlined in section 3.5. In the next section we discuss our treatment of the special grammatical problems posed by replacements, deletions and  $L_p$  macro declarations.

## 5.5 Alternative modifications to $L_p$

### 5.5.1 Replacements

The macro replacement, with general form,

```
      replace <target class> rule <abbreviated template>
(5.33)      by <macro template>
            [where <assertion>]
            means <macro body> endef
```

replaces the  $L_P$  form, identified by the abbreviated template, with a new form, specified by the macro template, (optional) assertion and macro body, for subsequent definitions and the  $L_P$  program.

At macro definition time, structures are created for representing the new template, assertion and body in the same way as for a macro definition. We then create a (macro) record descriptor for replacing the (lb or macro) record which is referenced in the parsing table for describing the old form. But because the type 2 macro bodies of definitions coming before the replacement in the  $L_E$  definition file might contain text which corresponds to the old form, we cannot simply throw the old record away. Instead, we must scope the replacement definition to apply only to subsequent  $L_P$  text.

For this we use a global, called `current_level`, and two fields, `lev` and `previous` in the macro record descriptor (declared in (5.25)). We set `previous` in the new record to point to the old (lb or macro) descriptor. We then modify all entries in the parsing table, which point to the old descriptor, to point to the new descriptor. Hence a table entry, corresponding to a `targetclass` and incoming symbol, may point to a list of several descriptors, each of which describe a separate  $L_B$  or extended  $L_P$  form. Moreover, for each macro defined in the  $L_E$  file, we prescribe a new integer definition level, in increasing order for successive definitions. The global, `current_level` (initially 0) is used for this. Each time we introduce a new (macro) descriptor into the table (for either a definition or replacement), we increment the `current_level` by one and put that new level into the `lev` field of the new descriptor.



At call time, the appropriate descriptor is chosen from a table entry's list as follows. Beginning at the most recent descriptor, the list is searched for a descriptor whose lev field is less than the current\_level. Since current\_level is increased by one just before the  $L_P$  program is compiled, all parsing decisions made for the program are based on the most recently defined  $L_P$  form. For the expansion of a call to a type 2 macro, we stack the current\_level and reset it (from the lev field in the macro's descriptor) for reflecting the definition level of the macro being expanded. Hence parsing decisions made for compiling the expanded macro's source list (in step 4 of (5.31)) are based only on original  $L_B$  forms and definitions coming before the macro in the  $L_E$  definition file.

#### 5.5.2 Deletions

Similar to replacements, the deletions of  $L_P$  forms, specified in general by

(5.34) delete <target class> rule <abbreviated template> endef  
apply only to the compilation of subsequent  $L_P$  text.

The  $L_P$  rule which is to be deleted is identified by way of the table entry corresponding to the target class and one of the starting symbols computed from the abbreviated template. We make sure that the (lb or macro) descriptor identified by the abbreviated template exists and is unique; we, incidentally do the same for replacements. Once the descriptor has been identified we simply mark its del field to indicate that the corresponding  $L_P$  form is no longer applicable.

We have made the restriction that all deletions come at the end of the file for reasons of efficiency. Subsequent definitions might have introduced templates with starters shared by several deleted templates; this would have required constructing a complex tree structure for implementing the search process (described in the next section) for taking account of definition levels. This structure is not so



difficult to implement but requires greater (memory) space and execution time; in fact we have had to use such a structure for implementing the macro declaration which we describe in the next section. In any event, we feel the present facilities presented for defining, replacing and, finally, deleting  $L_P$  forms is flexible enough for a super user who is modifying the original  $L_B$  compiler for implementing his extended language  $L_P$ .

Because all deletions come at the end of the file, they apply only to the subsequent  $L_P$  program. The decision procedure used by compile for deciding which (original  $L_B$  or macro) definition is to be invoked for a target class and incoming symbol is declared in the Algol W program as follows.

```

procedure decide (integer value target);
comment--Given incoming symbol, choose rule for satisfying target class;
begin
    rule := if symbol = "$class" then table (target, classrep)
           else table (target, indexed (symbol));
(5.35) while rule is macro and lev(rule) >= current_level do
    rule := previous (rule);
    if current_level >= program_level
    and ((rule is macro and del (rule)) or
        (rule is lb and lbdel (rule))) then rule := null
end

```

### 5.5.3 Macro declarations

Since a macro declaration, with general form (4.6), must appear in the text of the  $L_P$  program (and not nested within another definition) the  $L_B$  procedure which processes it first checks to make sure input (the global associated with lexical analysis) is set to true. The macro declaration is recognised like any other macro definition; internal structures are created for its template, (optional) assertion and macro body. Intype 1 macro bodies tagged with eval global names are bound according to their declarations in the current  $L_P$  environment. Finally, a procedure is invoked for modifying the grammar by way of the parsing

table.

Recall that the scope of a macro declaration is that block in which it was declared; therefore, the macro must be removed from the grammar upon exiting the block. Further, the declaration may temporarily replace several  $L_P$  rules corresponding to the target class for which the macro was declared; our process must allow for the restoration of all such rules when the macro is removed. Finally, we must make sure that no parsing conflicts, which are not local to the target class, are introduced for other syntactic classes. These considerations plus the fact that we must take account of deleted  $L_P$  forms complicate the process of modifying the parsing table for a macro declaration.

The procedure for modifying the table is as follows.

1. We increment `current_level` to obtain a new definition level for our macro.
2. We compute a set of possible starters (tokens and syntactic classes) for the target class (before modification) and a set of possible starters for the new macro template. Deleted forms are taken into account here.
3. We perform steps (a) and (b) for each starter in the set for the new macro template.
  - (a) We create a new (macro) record descriptor, containing the current definition level and pointers to the template, assertion and body. The previous field is pointed to the descriptor (if any) for the parsing table entry corresponding to the target class and starter symbol. This table entry is then  
(5.36) modified to point to our new descriptor.
  - (b) We perform steps (i) and (ii) for each syntactic class for which our target class is a starter.
    - (i) If the table entry corresponding to the syntactic class

and starter contains an (undeleted) descriptor which is not related to our target class (determined by way of the starter set for the target class) then a global parsing conflict has occurred. The conflict is reported and the entry is unmodified.

(ii) Otherwise, the table entry is modified to reflect the closure.

4. Finally, we update the environment list for the current block (normally used for declared names) with a record containing the current definition level (contained in the lev fields of the descriptors created in step 3).

Because we have created a separate (macro) descriptor record for each starter in step 3, all  $L_p$  rules (and so all corresponding descriptor entries in the table) which are replaced by our macro may be restored upon a block exit. Of course, we need not have created a separate descriptor for each starter since more than one corresponding table entry might point to the same rule descriptor. But we found creating separate descriptors for each starter was faster (in execution time) and required less code than a procedure for separating the starters into their relevant disjoint subsets.

Having modified our grammar by (5.36), calls to macro declarations occurring in subsequent  $L_p$  text in the block may be recognised by the decision procedure (5.35). The linear search based on definition levels works even for macro declarations because these macros may be physically removed from the descriptor lists upon block exit.

Removing the macro descriptors upon leaving a block is simplified by the lev fields we set in step 3(a) of (5.36) at declaration time. We just sequence through the environment list (starting with the most recently defined link) looking for records indicating macro declarations. Upon finding each such link, we restore all table entries which point to (macro) descriptors of the current definition level to point to their

old descriptors (by way of the previous field) and then decrement the current definition level by one. On completing the sequence, both the parsing table and the current definition level retain the values they had at block entry.

## 5.6 Generic declarations

Aside from the macro declaration, the generic declaration with form,

(5.37) declare <identifier> as <formal parameter ref>

is the only  $L_B$  construct involving a reference to the extension language  $L_E$ . Like any  $L_B$  form, (5.37) may be recognised in, and compiled from, either the  $L_P$  text of a macro body at definition time or the source list constructed for a 'flattened' type 2 macro body at the call time.

In the former instance, a special semantic procedure call is copied to the computation tree being constructed for the macro body. At call time this procedure call is invoked (without reference to the processor mode) for obtaining the type of the actual parameter matching the formal parameter referenced in the construct. At this point, semantic routines which perform the actual declaration are either copied across to a computation tree being constructed for a nested call if in parsing mode or executed immediately otherwise.

In the latter instance, where we are actually expanding a macro call, the appropriate actual parameter is queried for its type and the semantic declaration routines are either copied across to a computation tree or executed immediately depending on the mode in which the call is expanded.

An identifier declared in a generic declaration is initialised to either 0, false, "", or null as determined by the type of the actual parameter supplied in the call. All four constants have the same internal representation.

## 5.7 Summary

This concludes our discussion on the system's implementation. The  $L_E$  processor and  $L_B$  compiler both involve a substantial amount of recursion in the algorithms used and data structures defined. This reflects the structures of  $L_E$  and  $L_B$  rather than any personal taste for the heavy use of recursion.

As an implementation language, Algol W proved fairly satisfactory. Constructs we missed include compile time constant names, procedure pointers (which could be assigned to variables) and a larger number of record classes; because Algol W restricts one to 15 record formats, one is always trying to share a single record among various abstract objects. Facilities in Algol W which we heavily used and much appreciated include recursion, a good language for expression objects, eg conditional expressions, and good run time diagnostics.

One drawback to the processor's implementation is its sheer size. The Algol W compiler produces approximately 95K bytes of code for about 1450 Algol W statements. Compiling a respectable  $L_E$  program requires about 200K bytes of memory on the IBM 360/44 at St. Andrews. Exact timings were difficult to obtain because of the nature of the system's interpretation of the clock; this is probably just as well since the code produced by the Algol W compiler is not the most efficient. In any case, our primary purpose in implementing our processor was to demonstrate that our syntactic macro facility is implementable.

## CHAPTER 6

### Topics Related to the Syntactic Macro

In this chapter, we firstly discuss three features which might be included in a syntactic macro facility but which we have chosen to exclude from our extension language: recursive macro definitions, nested macro definitions and a more general form for specifying the base language  $L_B$ . Secondly, we discuss the possibility of basing our extension mechanism on a (bottom up) LR class of grammars.

#### 6.1 Recursive macro definitions

We might have designed  $L_E$  so as to permit a recursive macro definition analogous to those permitted in more general purpose macro processors, eg TRAC (Mooers, 1966). In terms of our syntactic macro definitions, this would permit reference to the new  $L_P$  form being introduced by the definition in the macro body. Such a facility would permit an alternative means for handling repetitions over a list, prescribed in the macro template by way of a list subtemplate. Recursive definitions would have to be restricted to type 2 definitions, where syntactic analysis of the macro body is deferred until call time.

In fact, our present implementation of the type 2 macro definition would require little modification for permitting recursion. A recursive type 2 macro definition for introducing an  $L_P$  construct for summing a list of expressions might take the form illustrated below.

```

define $expression
  rule 'sum' $expression1 ',' $expression2
    others: (* ',' $expression *)
  means
    given others
(6.1)    then list
            [($expression1) + (sum $expression2),
            forall others: [, $expression.others],[)]
            end
          else [($expression1) + ($expression2)]
        endef

```

Calls to (6.1) must have at least two expression parameters; calls containing more than two parameters will cause a recursive call to (6.1). Permitting recursive definitions such as this would require our modifying a single condition in the source text of our implementation. As it stands procedure decide (5.35) restricts the definitional scope of a macro to subsequent macro definitions (and the  $L_p$  program). The condition, "lev(rule) >= current\_level", in (5.35) could be changed to "lev(rule) > current\_level" so as to permit a macro definition to apply to itself as well as to subsequent definitions.

We have chosen to prohibit recursive definitions in  $L_E$  because  $L_E$ 's purpose is not only for extending  $L_B$  to a new language  $L_p$ , but also for producing a compiler for  $L_p$ . It is our opinion that a compiler should at least be guaranteed to halt, even for erroneous source programs. If we were to allow recursive definitions, ill-formed definitions (which cannot be detected at definition time) might cause certain calls to loop indefinitely at compilation time.

## 6.2 Nested macro definitions

As Leavenworth suggested in his proposal, a syntactic macro definition might have macro definitions nested in its defining body. Such nested definitions might have one of two interpretations; the first assumes that the nested definitions are local to the outer macro



definition and the second assumes that they apply globally upon a call to the outer macro.

The first interpretation is like that for procedure declarations nested in a procedure body in an Algol-like language program. Here, a nested macro definition is scoped to apply only to the outer macro's defining body. Nested macros with this interpretation can facilitate the writing of complex macro bodies. For example, assume we want to extend  $L_B$  to include a factorial operator like that introduced by (3.47) but without introducing a for statement. The macro definition might make use of a nested macro for introducing a for statement for the duration of the outer definition as follows.

```

define $factor
  rule 'factorial' $factor1.int
means
  define $statement
    rule 'for' $identifier ':=' $expression1.int
      'to' $expression2.int 'do' $statement
  (6.2) means (body as in 3.46) endef;

  eval:[ begin
    let fact = 1;
    for X := 2 to $factor1do
      fact := fact * X
    → fact
  end]
endef
```

The for macro, nested in (6.2), applies only to the defining body of (6.2) and has no global effect on  $L_P$ . This interpretation is similar to that for macro declarations; macro declarations are local to a specific block in the  $L_P$  program but nested macro definitions are local to a specific macro body.



The global interpretation for nested macro definitions is more interesting since a single macro call may invoke several further definitions. This interpretation has been used in earlier macro processors, eg by McIlroy (1960), in ML/I (Brown, 1967) and in the Schuman and Jorrand proposals for the syntactic macro (1970). In terms of our facility, a single call could radically change the structure of  $L_P$  in the midst of an  $L_P$  program. Consider the following outline of a macro containing nested definitions (indicated by subscripted d's).

```
define <target class>
  rule <macro template>
(6.3) means
       $d_1; d_2; \dots d_n;$ 
      [ $L_P$  text]
endef
```

A call to (6.3) would not only expand to the  $L_P$  text in the macro body but would also invoke further macro definitions (and so introduce additional  $L_P$  forms) corresponding to  $d_1, d_2, \dots$ , and  $d_n$ . The definitions invoked on successive calls to (6.3) need not be equivalent since the definitions for  $d_1, d_2, \dots$ , and  $d_n$  may themselves depend on the actual parameters supplied to such calls. In Appendix 1, we give an example of such a definition from Schuman and Jorrand for introducing a new data type into a language; calls to this definition not only expand to allocate space for new variables but also define new operators for the variables.

Although both (the local and the global) interpretations of nested macro definitions have their merits, we have chosen to exclude nested macro definitions from  $L_E$  altogether. Our reasons for this differ for each interpretation.

Firstly, we do not feel that the definitional flexibility given by nested local macro definitions is great enough to justify the complexity in interpreting the nested macros. Unfortunately, representing nested environments of local  $L_P$  rules (by way of a deterministic

parsing table) is a great deal more complicated than representing nested procedure declarations for an Algol-like language. Like relaxing the restriction that deletions come at the end of the definition file (section 5.5.2), nested local definitions complicate both the maintenance of the parsing table and the parsing decision algorithm.

We have excluded nested global definitions for reasons of transparency, the detrimental effect they would have upon error diagnosis and the complexity of their implementation. We believe that  $L_P$ , defined by the super user in the definition file, should remain fixed throughout the end user's program. Any local modifications made to  $L_P$  by the user should be transparent; the macro declarations he may make follow definite lexical scope rules. On the other hand, nested global definitions would permit the introduction of several 'hidden'  $L_P$  rules by a single macro call; this is further complicated by the fact that  $d_1, d_2, \dots$ , and  $d_n$  in (6.3) might be macro replacements or even macro deletions. Further, the definitional power of nested global definitions in the proposal of Schuman and Jorrand is dependent on a pure text replacement interpretation to the syntactic macro; this goes against our desire for tokenisation of the source text and reliable syntactic error diagnosis. Jorrand himself has later admitted (1976) that the 1970 proposal was too ambitious to be of practical use. Finally, nesting global macro definitions in a macro body blurs the distinction between  $L_B$  and  $L_E$ . Although treating  $L_E$  as a subset of  $L_B$  fits nicely with Leavenworth's original proposal, such treatment would necessarily complicate the implementation of the facility as a whole. We must remember that our goal is not to define a, purely textual, general purpose macro processor but a facility by which the user may extend a base language  $L_B$  to a new language  $L_P$  and end up with a practical compiler for the new  $L_P$ .

### 6.3 Generalising $L_B$

In an earlier paper (Campbell, 1978), we proposed a facility whereby the user might choose a particular base language  $L_B$  from several base languages represented on an external file. He could then extend the chosen  $L_B$  to define a new  $L_P$  which might then be saved on the same external file. The syntax of an  $L_E$  program for such a system might be as follows.

```
select < $L_B$  name>
(6.4) [< $L_E$  definition file>]
      [program < $L_P$  program> endprogram]
      [save < $L_P$  name>]
```

In (6.4), the  $L_B$  name and  $L_P$  name specify the external file names by which internal representations of the languages are referenced, the  $L_E$  definition file is as defined in section 4.1 and the square brackets ([,]) indicate that the constructs they enclose are optional.

The concept of a generalised base language is not new. For example, McIlroy (1960) proposed a more text orientated macro processor for compiler extension which is applicable to several base languages. And ECT, an extensible and contractible translator (Solntseff and Yezerski, 1972) allows the user to extend one of several base languages stored on a file, although the meanings of these extensions must be expressed in a separate systems programming language. In general, permitting the user to choose from among several base languages acknowledges the fact that there exists no universal base language from which one may effectively define any extended language  $L_P$ . Generality for defining new  $L_P$ 's is instead provided by generality in choosing  $L_B$ .

Unfortunately, although this facility is invaluable in theory, it is not so easily implemented in practice and it does not fit into our definition of the syntactic macro facility.

Representing the context free syntaxes of several alternative base languages is not so difficult so long as each base language has a grammar conforming to our LL(1) conditions.  $L_B$  forms might be represented by the list structures we use for representing  $L_P$  macro templates. Further, purely constructive semantics which are constants of the context free syntax could be represented on a file using code trees. Of course, one would require some software for generating these initial  $L_B$  representations.

The difficulty in implementing a generalised  $L_B$  arises in respect to the analytic semantics of the various base languages. For our facility, there is a strong connection between  $L_B$  and the extension language  $L_E$  in regard to analytic semantics so that type checking may be carried through to the extensions. If we were to allow various alternative base languages then we must cater for all sorts of context sensitive restrictions on those languages in our type checking facilities in  $L_E$ . The  $L_E$  we have defined in the preceding chapters reflects the specific compile time type restrictions of our illustrative base language  $L_B$ . Certain  $L_E$  constructs such as macro declarations in  $L_P$  and generic declarations assume a specific meaning to scope and to types of names. If we had designed  $L_E$  to be completely independent of  $L_B$  then we would not have been able to carry context sensitive error diagnostics for  $L_B$  through to the extended  $L_P$  in the way we have described in preceding chapters.

This is not to say our syntactic macro facility is, in general, dependent upon our chosen illustrative base language,  $L_B$ . As stated in section 5.3, our particular implementation is easily modifiable for applying  $L_E$  to one of many base languages whose syntaxes can be described by LL(1) grammars. Doing so involves writing new  $L_B$  semantic routines, modifying internal record definitions for representing the compile time types for the new  $L_B$  and, if necessary, restructuring the internal environment list for reflecting alternative scope rules. In principle,

our syntactic macro facility is applicable to, at least, most Algol-like programming languages.

#### 6.4 Syntactic macros in an LR parse

Although we chose to follow Leavenworth in embedding our syntactic macros in a deterministic top down parse, similar to that for LL(1) grammars, there is no intrinsic reason why they cannot be fitted into a bottom-up (LR) parse. In fact, Cheatham (1966) proposed that his syntactic macros be embedded in a simple precedence scheme. We chose a top down parse since alternative syntaxes are so easily specified in the macro templates (using subtemplates).

Although simple precedence grammars (Wirth and Weber, 1966) require choppy, and so unmeaningful, production rules (for avoiding precedence conflicts), there exist alternative families of LR grammars which permit greater flexibility in defining production rules, thus permitting more meaningful macro templates. For example, the LALR (Lookahead LR) grammars, described by Aho and Ullman (1973) and used in the YACC compiler-compiler (Johnson, 1975), permit a great deal of flexibility in the specification of production rules. In theory, LR grammars can describe a larger class of languages than can LL grammars. An important point is that the processor must be designed so as to notify the user of any ambiguous  $L_P$  forms introduced by macro definitions.

Of course, fitting the syntactic macro into an LR context would require a new parsing algorithm and alternative algorithms for modifying the LR parsing decision table to reflect macro extensions. Subtemplates might be translated (internally) to special nonterminal symbols which are defined by production rules corresponding to the nested templates. One would also want to redefine the description of  $L_E$  so as to reflect the bottom up nature of the underlying parse; eg the term, target class, might be described in terms of a reduction class. Similar, if not

equivalent, context free and context sensitive restrictions may be applied to macro parameters for providing the same error diagnostics provided in the LL parse.

Although embedding our extension facility in an LR parse would require a major redefinition of  $L_E$  and the rewriting of its implementation, the results might reward the effort. For example, we might find that an LR parse permits greater flexibility than does the LL parse for introducing new infix operators while retaining the flexibility for defining the more prefix orientated  $L_P$  forms.

In the next, and final, chapter we draw some conclusions on the facility we have defined in preceding chapters and suggest areas for further investigation.

## CHAPTER 7

### Conclusion

Our overall objective has been to make the syntactic macro more effective for extending high level languages. In the following sections we list the contributions which we feel we have made toward this objective, we suggest areas for further investigation and, finally, we say something about syntactic extensibility in general.

#### 7.1 Contributions

The most important contributions we have made toward improving the effectiveness of the syntactic macro are as follows.

1. We have defined a more flexible notation for specifying alternative syntaxes in the macro template. Our notation permits more variety in the kinds of  $L_P$  forms which may be introduced than that permitted by either Leavenworth (1966) or Vidart (1974). Macro calls need not begin with a delimiting token. Recursively defined sub-templates permit a notation similar to popular extended forms of BNF, eg Wirth's proposal (1977). The restriction that no template can represent the empty string is a minor one since emptiness can be expressed using optional subtemplates.
2. The non-procedural language for prescribing text replacement in the macro body is simple and reflects the nested notation used for denoting alternative forms in the macro template. Each  $L_E$  construct in the body reflects either a specific subtemplate or some context sensitive (type) condition upon which text is expanded. Further, the names of formal parameters and subtemplates are scoped within corresponding  $L_E$  constructs for prohibiting references to parameters and subtemplates which are not matched at call time.
3.  $L_P$  forms may be replaced or deleted. Extensibility implies not only an ability for expanding  $L_P$  but also for pruning  $L_P$ . Replacements and deletions follow the same grammatical restrictions imposed on macro definitions.



4. Macro declarations are given to the  $L_P$  end user. The scope of a macro declaration extends to the end of the block in which it is declared. The new  $L_P$  form it introduces cannot conflict with old forms but takes precedence over old (global)  $L_P$  forms. Further, free variables may be used in the macro body. Since the scope of both a macro declaration and free variable names in its body is determined lexically, its use is all the more transparent to the user.
5. Our most important contribution is in carrying the error diagnostic capabilities of the base language  $L_B$  through to the extended language  $L_P$ .

Since parameters are restricted to specific syntactic classes, actual parameters can be diagnosed for syntax errors at recognition time. All type 1 macro bodies and all syntactically tagged text segments in type 2 macro bodies can be diagnosed for syntax errors at macro definition time. In all of these cases syntax errors may be flagged where they appear in the source text and syntax error recovery is as good as it is for  $L_B$ . The only place syntax error diagnostics become blurred is in the expansion of type 2 macro calls; here, a warning of the nested nature of an error is issued to the user.

Type violations are flagged in the source text as early as possible. Type violations in actual parameters corresponding to formal parameters which have been tagged with types in the template are flagged precisely where they appear in the source text. More complicated type restrictions can be expressed using assertions in the where clause. The fact that the where clause is separate from the macro body clarifies the distinction between their respective functions. Type violations detected by the where clause are flagged immediately following the call in the source text. Finally, type 1 macro bodies which have been tagged with eval may be diagnosed for type violations as they are parsed at definition time. Unfortunately, it is impossible



to check types in unexpanded type 2 macro bodies because of the context sensitivity of types. When type violations are diagnosed at macro expansion time, a warning of their nest in the call is issued to the user. Of course, the user must be aware that typed parameters are passed by reference and that untyped parameters are passed by pure textual substitution.

It is the flexibility for conditional expansion in the type 2 macro which causes most trouble. But if the super user (who is defining  $L_p$ ) can be sure that his type 2 macros will expand correctly for legal calls, he may then impose restrictions on the parameters for rejecting illegal calls made by the  $L_p$  end user.

6. Finally, an advantage of our implementation is the relatively small overhead it imposes on the compilation of  $L_p$  programs. Firstly, as a compiler for (an unextended)  $L_B$ , our processor compares favourably with conventional syntax directed compilers. The processor compiles  $L_B$  text using the method of recursive descent but uses a parsing table for reducing target classes to appropriate production rules; this is an effective method for parsing any LL(1) language. Secondly, the start of a macro call, like the start of an  $L_B$  rule, is recognised against the parsing table. Recognising the remainder of a call (against the template's internal list structure), no matter how complex, requires no more effort than that required for recognising an  $L_B$  rule of similar complexity (using a recursive procedure). Admittedly, the repetitive rescanning of expanded type 2 macro calls does introduce increased overheads in time. But for those macros for which computation trees have been constructed at definition time, expansion requires no more effort than that required for generating code for  $L_B$  rules. The greatest overheads (in both time and space) are required for interpreting the macro definitions - not the calls. A user pays only for the macros he defines.

## 7.2 Possible modifications and areas for further investigation

There are two relatively minor modifications one might make to  $L_E$ .

Firstly, we might have used meta-symbols in the macro template for denoting alternative syntaxes for  $L_P$  forms which conform to the more popular notations for extended BNF, eg the proposal of Wirth (1977). For example, we might have used [ and ] in place of ( and ), and { and } in place of \* and \*. Unfortunately, these special brackets are not available on the IBM 029 keypunch on which we were dependent. A terminal based on the ASCII character set would have permitted the alternative notation. We could have used the more popular notation in preceding chapters but we thought that our illustrations should reflect the notation to which we were restricted.

Secondly, during a visit to St. Andrews in 1976, Dr Phillipe Jorrand suggested that we use simple identifiers for naming parameters in the macro template in the same way we use them for naming subtemplates. eg a template might take the form,

(7.1) 'while' condition: \$expression.bool 'do' action: \$statement

The names, condition and action, would then be used in the macro body for referencing the respective parameters. Admittedly, this use of names is consistent with the naming of subtemplates. Yet, the use of syntactic class names (eg \$expression) in the macro body makes it easier for the user to assure himself that his replacement text is syntactically correct. Although we prefer this latter notation, a possible compromise would be to allow both notations.

In addition to these minor points, we see three characteristics in our facility which might be improved upon with further investigation.

Firstly, although our macros permit the introduction of  $L_P$  forms beginning with either nonterminals or subtemplates, the underlying LL grammar still imposes a prefix orientation on  $L_P$  forms. Defining  $L_P$

rules involving infix operators, especially where repetitions over a list of operands is desired, is more difficult than those involving prefix operators. Perhaps a facility which is embedded in an LR class of grammars would correct this deficiency. It might pay to further investigate the LR strategy we discussed in 6.4.

Secondly, the forall construct, used for expressing text replacement in the macro body, might be extended to permit more expressive power than is currently permitted. Consider its current form.

(7.2) forall 1 : <text>

Recall, 1 is the name of a list subtemplate in the macro template. If the subtemplate 1 is matched  $m$  times then the replacement text expressed by (7.2) is a concatenation of  $m$  text segments; the first segment corresponds to the first match of 1, the second segment corresponds to the second match of 1, and so on up until the  $m^{\text{th}}$  match of 1. As it turns out, this strict interpretation of the forall construct is useful for expressing the meanings of many  $L_P$  forms containing lists. Our object in defining all of the  $L_E$  constructs for expressing conditional text replacement in the body was twofold; we wanted a notation which was both transparent and relatively descriptive. Yet there is room for greater expression in the forall construct. For example, one may wish to index specific matches of a list subtemplate or to change the ordering of a list of parameters when prescribing replacement text. We proposed a procedural language for doing this (Campbell, 1978) but that language was not nearly so transparent as the non-procedural language which we have defined here. There is scope for further examining the sort of text replacement one might wish to express for repetitions over a list and for defining a non-procedural notation for expressing it. The iterators (based on quantifiers) which Earley (1974) has proposed for describing ordered sets of abstract objects might form the basis of such a notation.

Thirdly, if our syntactic macro facility is to be truly useful, it must be defined on top of a more practical general purpose base language containing procedures, functions and arrays.  $L_B$ , as we have defined it, was chosen for illustrating  $L_E$  rather than for use as a general purpose programming language. One might choose an existing language for  $L_B$ ; Pascal (Wirth, 1971) is an obvious choice in the LL context. There would be size problems if the processor for this is rewritten in Algol W; a better implementation language might be Pascal, Algol-R (Morrison, 1978) or a systems implementation language. If our implementation strategy were to be followed then the implementation language would certainly require structures, pointers and minimal string handling facilities. Yet garbage collection facilities would not be required since all lists constructed for macro definitions remain throughout the compilation process. The only time at which list space must be reclaimed is during macro expansion; since the expansion process is of a nested nature, its lists could be organised on a retractable stack. The advantage of choosing a language like Pascal for  $L_B$  is its popularity; this would in turn create a wider market for the syntactic macro itself.

### 7.3 Syntactic extensibility

Although data type extensibility may be found in many currently popular high level languages, eg mode indications in Algol 68 (van Wijngaarden et al, 1975) and type declarations in Pascal (Wirth, 1971), syntactic extensibility has not met with the same success. Languages which permit syntactic extensibility are rarely used except by those people who designed them and their students. T A Standish (1975) has written an eloquent account of the initially high expectations held for extensibility in the 1960's and the subsequent disappointments experienced in the 1970's.

We feel syntactic extensibility has failed to make an impact on programming language design for three principal reasons. Firstly, the transparency of paraphrase facilities, whereby new linguistic forms may be defined in terms of existing forms, is marred by a failure to carry the error diagnostics of the old forms through to the new forms. Secondly, most systems for syntactic extensibility have been defined on top of newly designed base languages or corrupted versions of existing languages;  $L_B$ , as it stands, puts our facility in this category. The user's ability to express new  $L_P$  forms in terms of 'familiar'  $L_B$  forms is diminished by his having to familiarise himself with the new  $L_B$ . Thirdly, the exaggerated claims made for extensibility in the 1960's (recounted by Standish) have blinded a disappointed programming community to the more modest advantages which syntactic extensibility has in programming language design.

We feel we have made a significant contribution on the first point; we have summarised the error diagnosing capabilities of our facility in section 7.1. On the second point, designers probably chose to base their systems on newly defined base languages either in an ambitious attempt to define a 'universal' base language from which diverse languages could be defined, or, like ourselves, for illustrating the salient properties of their extension mechanisms. Our suggestion in section 7.2, that this problem could be solved by defining  $L_E$  on top of a more popular existing language like Pascal brings us to the third point. Extensibility was introduced amidst much euphoria about its being the solution to the problem of a multiplicity of programming languages. The idea was that a universal base language together with a set of powerful but simple extension mechanisms could form a system whereby personalised languages could be manufactured by relatively unsophisticated programmers at will. This initial euphoria was soon dispelled; no such base language could be found and the extension mechanisms were either too complicated to use or simply not powerful

enough for introducing the diverse extensions desired. Since then, extensible language designers have been more modest.

We choose to follow Standish (1975) in viewing syntactic extensibility, and the syntactic macro in particular, simply as a programming tool which might be made available to the general purpose language programmer. Just as the programmer has facilities in his current high level languages for declaring variable names and data structures (types) peculiar to his applications, one might give him the syntactic macro as a tool whereby he may write expressions and commands in a notation conforming to his particular application area. As its name implies, the syntactic macro is a tool for modifying the syntax of expressible forms; no semantic extensibility is claimed since expressions in the new  $L_p$  may be written using the original  $L_B$ . The syntactic macro which we have described in preceding chapters is well suited to such an interpretation. As we have said in section 7.1 the syntactic macro may be implemented on top of an existing programming language without significantly increasing the overhead of compilation for that language. The user has the syntactic macro available to him when he wants it and he pays little or nothing when he does not wish to use it.



## APPENDIX 1

### Previous Work Concerning the Syntactic Macro

Language extensibility, in general, is surveyed in a paper by Solntseff and Yezerski (1974). General purpose macro processors are surveyed in a paper by Brown (1969) and, more recently, in two textbooks by Cole (1976) and Brown (1974). Our survey covers some proposals made for the syntactic macro itself and some of the languages which use the syntactic macro (in one form or another) for achieving syntactic extensibility. Firstly, we look at an earlier proposal for extending high level languages by using macros.

#### A1.1 McIlroy's macros

McIlroy (1960) proposed that some of the macro definitional facilities found in assembly languages be applied to high level languages like Algol. New linguistic forms would be defined in terms of old forms but the new forms (ie the macro calls) would be recognised by delimiting symbols, possibly new basic symbols introduced in the extensions. Macro expansion is done at the lexical level. Some of the features of McIlroy's macro follow.

1. The source text is tokenised.
2. Base language statements have macro time counterparts for controlling conditional macro expansion. For example a list of parameters to a call might be handled by a macro time for statement in Algol.
3. Calls and global macro definitions may be nested in other macro definitions.
4. The facility is applicable to a variety of base languages since macro expansion is performed at the lexical level.

Unfortunately, as with any text processor, McIlroy's macros can be inefficient, depend on delimiting symbols and fail to carry the error diagnostics for the base language through to the extended language.

A more recent application of this kind of macro is found in the extensible language, McAlgol, designed by Bowlden (1971). McAlgol is essentially a preprocessor to Algol 60. Bowlden notes its poor error diagnostics and the user's tendency to make corrections to the intermediate (Algol 60) text rather than to the (McAlgol) source text.

## A1.2 The syntactic macro facility

As stated in Chapter 1, syntactic macros were first introduced in two separate papers by Cheatham (1966) and Leavenworth (1966). Having discussed Leavenworth's proposal in Chapter 1, we shall first discuss Cheatham's macros and then look at two alternative syntactic macro facilities proposed by Schuman and Jorrand (1970) and by Hammer (1971).

### A1.2.1 Cheatham's macros

Cheatham (1966) actually introduces three different kinds of macros for extending high level languages: lexical macros, syntactic macros and computational macros.

Cheatham's syntactic macro facility (called an smacro) is similar to Leavenworth's but is designed to fit into a simple precedence (bottom up) grammar. The following smacro definition from Cheatham illustrates the general form; we have made changes to the lexical notation for readability.

```
      let n be <integer>
(A1.1)  smacro matrix (n) as <attribute>
        means 'array (1:n,1:n)'
```

This smacro definition introduces a new BNF rule for <attribute> (Cheatham's term for <declaration>),

```
(A1.2)  <attribute> ::= matrix (<integer>)
```

whose meaning is given by the quoted text in (A1.1). Calls to (A1.1) are recognised according to the modified simple precedence grammar



and the text matching parameter *n* must satisfy the syntax of an *<integer>*. The entire call reduces, syntactically, to the nonterminal *<attribute>*. Macro expansion is just like that for Leavenworth's macros; eg

(A1.3) 'matrix (25) *i*' expands to 'array (1:25,1:25) *i*'

Cheatham's smacros are less powerful than Leavenworth's in that they allow neither alternative syntaxes in the macro template, nor conditional expansion in the macro body. On the other hand, a smacro may be used for introducing an alternative form for any nonterminal in the underlying grammar where one may introduce only new statement and primary forms using Leavenworth's macros.

Lexical macros are similar to smacros but calls are recognised by the special delimiting token *%*. This symbol takes precedence over any other symbol for triggering expansion of a macro call. Taking another example from Cheatham,

(A1.4) let *n* be *<integer>*  
macro matrix (*n*) means 'array (1:*n*,1:*n*)'

introduces the BNF rule,

(A1.5) *<macro>* ::= *% matrix* (*<integer>*)

Calls to (A1.4) may appear anywhere in the subsequent source text. The advantage of overriding the usual syntactic constraints is the lexical macro's main disadvantage; it makes nonsense of the syntactic macro.

The computational macro is Cheatham's answer to the rescanning problem for string replacement in the smacro. Its body is written in a low level (assembly) language so that the body need not be re-compiled for each call and so that replacement code may be tuned. Of course, this forces the user to write his own code trees.

Cheatham also defines special system macros for performing tasks which are not easily expressed by the three kinds of macro mentioned above. Although these allow the user to express semantic operations not allowed in the simpler Leavenworth notation, they (together with computational macros) demand extensive knowledge of the underlying implementation.

### A1.2.2 The Schuman and Jorrand Proposal

Schuman and Jorrand (1971) have published a report in which they examine both data type extensibility and syntactic extensibility. The latter, which is our interest here, is examined in terms of a model which is an ambitious extension to the Leavenworth model.

The authors' syntactic macro definition mechanism consists of three parts: the production, where the new syntactic rule is given; the predicate, which imposes certain additional conditions which must be satisfied by a macro call; and the replacement, describing that text which is to replace the macro call. Since we have already seen examples of productions (macro templates) and replacements (macro bodies) we shall look more closely at the predicate. But, firstly we shall look at the authors' notion of the nested global macro definition. Because their model defines the macro definition to be an integral part of the base language, a macro body may contain definitions which come into effect upon a call to the outer macro. Consider an example from Schuman and Jorrand; uppercase names denote syntactic classes (nonterminals).

```

macro DECL0 ::= 'TYPE1 stack ( EXPR1 ) IDEN1;'
      means
      'TYPE1 array (1:EXPR1) IDEN1;
      integer level_IDEN1 initial 0;
      macro PRIM0 ::= 'depth_IDEN1'
            means 'result(EXPR1)';
      macro PRIM1 ::= 'IDEN1'
            means
(A1.6)      '(if level_IDEN1 gt 0 then
              (IDEN1 (level_IDEN1),
               level_IDEN1 := level_IDEN1 - 1;)
              else error("overflow IDEN1"))';
      macro REFR0 ::= 'IDEN1'
            means
              '(if level_IDEN1 lt depth_IDEN1 then
                (level_IDEN1 := level_IDEN1 + 1;
                 IDEN1 (level_IDEN1))
                else error ("overflow IDEN1"))';

```

This example makes several assumptions, one of which is that the definition does not conflict (syntactically) with the underlying grammar. But since we are dealing only with a model here, let us look at some of the points illustrated by (A1.6).

1. The purpose of the macro definition is to introduce a push down stack as a new structured data type.
2. Each macro call produces not only two declarations but also three additional production rules to the underlying grammar. In general, it may be possible to drastically change the grammar with a single macro call.
3. The example assumes a literal string replacement; tokens may be concatenated together to form new tokens, eg `level_IDEN1`. Further, what were once names can become reserved symbols upon a macro call.
4. There is a bit of fiddling with the base language. For example, the then arm of the second nested definitions assumes one can have a statement which decrements the array index after returning the value of an array element.

In general, the power of expression implied for the model by (A1.6) precludes any security; eg no syntax analysis can be performed in the body at definition time nor can the user be sure of what the language is throughout his program. We must remember that the authors' aim was not to define a practical tool but a model with which they could examine the limits of extensibility.

The predicate, used for further qualifying macro calls, is an addition to the Leavenworth model. The example which follows comes from Schuman and Jorrand.

```
(A1.7) macro FACTOR0 ::= 'PRIM1!'  
      where is_integer ( PRIM1 )  
      means 'factorial ( PRIM1 )'
```

Here, is\_integer is a primitive predicate which returns the value true if its operand corresponds to an actual parameter of integer type. Predicates can be more complicated expressions built from primitive predicates and the operators of the predicate calculus. The authors also propose predicates which query the syntactic structure of an actual parameter. For example, the predicate

(A1.8)  $\text{PRIM}_1 \Rightarrow '( \text{EXPR} )'$

would yield true if the actual parameter matching  $\text{PRIM}_1$  grammatically produced (or was of the structure),

(A1.9)  $( \text{EXPR} )$

Such predicates might be useful for producing optimised code at macro expansion time.

In general, Schuman and Jorrand see the power of the syntactic macro as being dependent on the power of these predicates. On the other hand, predicates such as (A1.8) could prove difficult to implement. During a visit to St. Andrews in 1976, Jorrand admitted that their proposal, in general, was much too ambitious and ambiguous to be implemented as it stood. Yet their model does push extensibility to its limits and their predicate is the basis of our (more subdued) assertion clause.

#### A1.2.3 Hammer's alternative to the string replacement process

Leavenworth, Cheatham, and Schuman and Jorrand all assume a string replacement quality to macro expansion. But Hammer (1971) observed that the syntactic structure of every copy of a macro body is identical (given no conditional expansion) since corresponding actual parameters supplied at call time are of identical syntactic type. Consequently text in the macro body can be parsed at macro definition time. Further, Hammer points out that certain additional actions might be executed at macro definition time while others must be deferred until macro expansion time. The problem is then to distinguish between these two sets of actions.

To this end, Hammer separates these compilation actions into two classes:

1. constructive actions, eg code generation, which may be done at macro definition time, and
2. analytic actions, eg data type checking, which must be deferred until expansion time.

One may then parse and perform constructive actions for a macro body at definition time, thus building an uncompleted code tree (with hooks for actual parameters). All analytic actions must, according to Hammer, be deferred until expansion time.

This proposal makes two basic assumptions about the base language:

1. it can be parsed by context free techniques alone and
2. the code tree is a constant function of the syntax.

Many syntactically described languages satisfy the first condition, although some parsers require information about names from a symbol table. The second condition can be relaxed for constructing at least a skeleton code tree.

Although no implementation of this method of parsing is recorded by Hammer, he was able to separate the compilation actions of BASEL, a base language for another extensible system called ELF (Cheatham, Fischer and Jorrand, 1968), into their two respective classes.

In fact, following Hammer, we have been able to carry the process further in the implementation of our own processor, as described in Chapter 3.

### A1.3 Applications to syntactic extensibility

We now look at some programming languages which contain syntactic extension mechanisms which are based on, or are similar to, the syntactic macro. The languages which we have chosen to examine either follow directly from the proposals discussed in the last section or have

peculiar characteristics which we find interesting. Although many of these also contain data type extension mechanisms, our interest is in their syntactic extensibility.

### A1.3.1 Definitions in Algol

Galler and Perlis (1967) proposed a system for making syntactic extensions to Algol 60 which is remarkably similar in effect to the syntactic macro. The base language, called Algol C, is a variant of Algol 60. By means of a definition mechanism, Algol C may be extended to many languages called 'Algol D's'. Each Algol D is the result of syntactic extensions to any of four syntactic entities in Algol C: <type>, <assignment statement>, <arithmetic expression> and <boolean expression>. For example, a new type, matrix, may be defined syntactically by

(A1.10) matrix (m,n) means array [1:m,1:n]

More complicated definitions may then follow for defining arithmetic and assignment operations on objects of type matrix. The authors give a set of such definitions in an appendix to their paper. The extension mechanism differs from the syntactic macro in that formal parameters, eg m and n in (A1.10), are not ascribed syntactic types and in the way macro calls are recognised and expanded. A macro call is recognised by a process of tree matching once the source text has been parsed; macro expansion is effected by syntax tree replacement.

An interesting aspect of the base language, Algol C, is that it contains an operator, loc, for obtaining the address of a variable; loc is like the l-value operator in BCPL (Richards, 1969). Although such an operator can be dangerous (since it may be used for overriding type rules), it is useful for optimising operations on multi-dimensional arrays. A definition set of matrix operations might use loc for short-cutting the expensive evaluation of subscripted variables. Since this operator would appear only in the definition, the Algol D user need not be aware of it.



### A1.3.2 ELF - an extensible language facility

ELF was designed by Cheatham, Fischer and Jorrand (1968) for making extensions to a base language called BASEL. The purpose of the system as a whole is to define a wide variety of problem orientated languages, each with its own data types and operators. To this end, BASEL is rich in mode constructors for defining complex aggregate modes in terms of the basic modes int, real, bool and char. Aggregates include rows (vectors), structures, procedures and tuples; a tuple is an ordered set of values which need not be of the same type. The user may also define operators on these types; this is where syntactic extensibility is required.

The authors chose to trim down the lexical macro and syntactic macro proposed by Cheatham (1966) for performing the simple task of introducing the syntaxes of new operators. In BASEL, an operator is simply a procedure plus some syntactic information on how a call to the procedure is to be written. The syntactic information is supplied using an operator declaration; eg

(A1.11) let ↑ be infixr prec >\*

introduces a new right associative infix operator ↑ whose precedence is greater than that of the \* operator. This information is used to modify the operator precedence table against which BASEL programs are parsed. The meaning of operators is given in terms of procedure values; eg

(A1.12) let + mean proc (bool a, bool b)  
                                  (if a then ¬b else b)

states that + is meaningful between two boolean values, and is to denote 'exclusive or' in that context. Calls are recognised in the context of the underlying operator parse and expanded to corresponding procedure calls. Although this is not a very rich use of the syntactic macro, the authors desired no more richness and so were right not to include it.

### A1.3.3 Vidart's extensions to GSL

Vidart (1974) employs a variation on Leavenworth's syntactic macro for defining extensions to GSL, the Grenoble Systems Language (Berthaud, Clauzel and Jacolin, 1972). GSL is a systems programming language which is described by an LL(1) grammar.

Vidart's syntactic macros are embedded in and directed by GSL's underlying grammar; for defining extensions this grammar is modified with the following extra BNF rules (substituting our own terminology for Vidart's).

```

    <statement> ::= <extension>
(A1.13) <extension> ::= extsyn <ext-rule> endext
    <ext-rule> ::= <target class> := <template> |-><body>

```

Extension statements invoke macro definitions for modifying the underlying grammar for introducing new linguistic forms. This modification is simplified by the restriction that each template (and so each call) begins with a basic token.

An improvement to Leavenworth's macro concerns the macro process. Since Vidart restricts parameters in the macro template to specific syntactic classes, the body may be parsed at definition time. This solves the problem of repeated scanning of replacement text and permits the detection of syntax errors where they appear in the body. Also, new linguistic forms may be defined for any syntactic class in GSL's grammar. On the other hand, Vidart permits no alternative syntaxes in the template nor conditional expansion in the body. Nor does he treat the more context sensitive (data type) qualities of the extended language. Generally speaking, our context free processing of the type 1 macro (discussed in section 2.2.1) is derived from Vidart's macro, but permits macro templates to begin with either a basic symbol or a syntactic class.

Vidart's contribution is essentially an (important) improvement to the processing of Leavenworth's macro and a demonstration that the syntactic macro can be applied to an existing language.



#### A1.3.4 ALEC - a language with an extensible compiler

ALEC was designed by Napper and Fisher (1976) to be a user extensible scientific programming language. Its base language is based on a subset of PL/I which is restricted to a small number of (numerical) data types. The system as a whole is written in (and embedded in) a revised version of the Compiler Compiler (Brooker and Morris, 1960), called RCC (Napper, 1973).

Syntactic extensibility in ALEC is provided for by two kinds of macros, informal macros and formal macros. The body of an informal macro is written in that language in which ALEC is implemented, ie RCC. This permits a sophisticated user, who has knowledge of RCC, to define meanings (of new forms) which cannot be expressed in ALEC. The body of a formal macro must be written in ALEC and resembles the body of a closed routine. Since ALEC itself is parsed against a non-deterministic top down grammar, the user of either type of macro must ensure that he does not introduce ambiguous forms. Our interest is in the formal macro since it more closely resembles the syntactic macro.

Formal macros may be used for introducing either new statement forms or new expression forms. Here, parameters are restricted to syntaxes representing values or variables; this is not the case with informal macros where more syntaxes are permitted. Although a formal macro is like a closed routine in this respect, the syntaxes of calls and the string replacement interpretation is similar to that for syntactic macros. For example, consider the following formal macro (where we again use our own lexical notation)

```
    open routine
      sumsq (real X) and (real Y) to (real variable tr subst Z);
(A1.14)      Z := (X * X) + (Y * Y);
    end
```

Parameters in the template are enclosed by parentheses. Notice that each parameter is declared to be of type real, but the last parameter is also tagged with 'variable tr subst'. This means that X and Y are expressions which are passed by value but Z is a variable which is passed textually. For example, the (statement) call

(A1.15) sumsq p + 1 and q + 2 to a(i)

would expand to the ALEC text

```
      begin; declare (X,Y) integer;  
(A1.16)    X = p+1; Y = q+2;  
            a(i) = (X * X) + (Y * Y);  
      end
```

ALEC permits other methods of passing parameters, eg 'by reference'. This variability in parameter passing is especially useful to a scientific applications language.

Another interesting feature of ALEC is the way it interprets the scope of names. Although all macro definitions must precede the program written in the extended language, bodies of the definitions may contain names declared in the program. Consider the following example given by Napper and Fisher.

```

open routine increment global array by (tr subst a);
  declare i integer;
  do i=1 to 10; s(i) = s(i)+a end;
end;

```

```

open routine add (tr subst b) and random number to
  global array;
  declare i integer;
  generate next random number i;
  increment global array by b+i;
end;

```

(A1.17)

```

begin program;
  declare (i, s(10)) integer;
  initialise s;
  :
  increment global array by 3;
  begin;
  :
  increment global array by 5;
  add i and random number to global array;
  :
  end;
end of program

```

A macro definition may refer to a (non-local) free name so long as every call to the macro is made within the lexical scope of the name. The binding of a name depends not only on where a name is declared but from where it is referenced. For this reason, ALEC makes a distinction between the static scope level and the dynamic scope level; the static level is incremented by one at each entry to a block and the dynamic level is incremented by one at each entry to a call. This two dimensional measure of scope may then be used in (A1.17) to distinguish among the *i* in the program call to add ..., the *i* in the body of add ... and the *i* in the body of increment ...; note that some confusion might have arisen since all parameter passing in the macro calls is by true (literal) substitution. Although ALEC's two dimensional measure of scope is strong enough to resolve all confusion, the use of deeply nested

names might confuse the user. Secondly, the definition of macros, containing free names, before the program assumes a great deal about the program itself, eg what names have been declared. It would be more transparent if, as Napper and Fisher suggest in their conclusion, macros were defined in the blocks where the names they required are declared.

Although our discussion of ALEC has concentrated on the formal macro, it should be said that greater flexibility is provided by informal macros both for specifying alternative syntaxes for macro calls and for specifying conditional expansion in the macro body by way of a procedural macro time language. The difference here is that the replacement text must be written in RCC rather than in the base language of ALEC.

#### A1.3.5 Syntactic extensions in BALM

Malcolm Harrison (1970) intended the name BALM (Block And List Manipulator) "to imply that its use should produce a soothing effect on the worried programmer". BALM has an Algol-like syntax but contains facilities for creating and manipulating lists, vectors, strings and functions as well as simple types. BALM programs are translated (by way of a bottom up precedence parse) to a Lisp-like intermediate code which is interpreted by the MBALM machine. The types of variable names are determined at run time so no compile time type checking is required.

A BALM program consists of a sequence of commands for creating and manipulating objects. For example, the following command sequence assigns numbers to X and Y, a function to sumsquares and prints out the value obtained by applying sumsquares to X and Y.

```
      X = 5;
(A1.18) Y = 6;
      sumsquares = proc (a,b), a↑2 + b↑2 end;
      print (sumsquares (X,Y))
```

Syntactic extensibility is provided for by facilities for introducing, deleting and redefining (infix or prefix) operators. Left and right precedences for these operators are specified absolutely using integer constants rather than in terms of the precedences of other operators. Meanings may be ascribed to operators either by associating them with functions or by the use of macros. For example, a sumsq operator might be introduced by the following operator definition.

(A1.18) infix ("sumsq, 1501, 1500, "sumsquares)

Here sumsq is defined to be an infix operator with left precedence 1501 and right precedence 1500 (associating to the left); these precedences are exactly those which are defined for the + operator. The meaning of sumsq is given by the function, sumsquares, which was defined in (A1.18). Therefore, the command

(A1.20) X sumsq Y

would be interpreted as a function call, sumsquares (X,Y). One can also define sumsq using a macro. This requires a minor modification in the operator definition and the use of the means command; eg

(A1.21) infix ("sumsq, 1501, 1500, "sumsq");  
a sumsq b means a<sup>2</sup> + b<sup>2</sup>

Subsequent commands of the form (A1.20) would expand to

(A1.22) X<sup>2</sup> + Y<sup>2</sup>

Operator definitions and macros may also be used for introducing more complicated statement forms. Although macros do not provide for conditional macro expansion, BALM contains facilities for querying the types of operands at run time for controlling execution and so providing for generic operators.

Source text is translated into intermediate MBALM machine code in two passes by way of a system procedure called TRANSLATE. Firstly, a (bottom up) precedence analysis pass parses BALM source text for producing a syntax tree containing operators, operands and macro calls.

A second pass traverses the tree (top down) for expanding nodes representing macro calls. The TRANSLATE procedure itself may be redefined by the user for providing yet another level of syntactic extensibility.

To summarise, BALM provides for extensibility in several ways; by user-defined functions, by operator definitions, by macros and by the use of a user-defined TRANSLATE procedure. Although the means command resembles a syntactic macro definition on the surface, macro recognition and expansion, unlike that for the syntactic macro, is performed after syntax analysis.

#### A1.3.6 ECT - an extensible contractible translator system

The ECT system, designed by Solntseff and Yezerski (1972) is a synthesis of compiler compiler and extensible language concepts. Using ECT, a user may implement a new base language from scratch, using a description language called METALANG or, alternatively, he may define extensions to one of several pre-prepared base languages.

The description language, METALANG, has three components:

1. a program restructuring language (PRL), for selecting and altering the translator tables of base language processors;
2. a syntax description language (SXL), for describing the syntax of a base language and for introducing syntactic extensions; and
3. a semantics description language (SML) for ascribing meanings to constructs in the base language and to any extensions. SML is an extension of McKeeman's XPL (1970) and is used for writing the semantic procedures corresponding to the BNF rules introduced by SXL.

The fact that ECT provides a separate systems programming language for describing semantic routines gives the user a choice of base languages to which he may make extensions (ie a generalised  $L_B$ ) and leads to efficient implementations of the extended languages. Yet, like any

compiler compiler model, this means that the user cannot express meanings in that language which he is extending but must grapple with another systems programming language.

An interesting feature of the PRL component is its mechanism for contracting a selected base language, by deleting  $L_B$  BNF rules. A similar mechanism for contracting (as well as extending) the syntax of a language is contained in Wegbreit's ECL programming system (1971).

Finally, the notion of synthesising compiler compiler and extensible language concepts is not unique to ECT. Standish uses compiler compiler mechanisms in conjunction with syntactic macros for defining syntactic extensions in PPL, a polymorphic programming language (1969), and Napper and Fisher rely heavily on the mechanism of RCC (1973) for defining extensions to ALEC (1976). As Standish (1975) observes, one of the earliest expressions of the extensible language concept was given by Brooker and Morris (1962) when they introduced the original Compiler Compiler:

"The system is extendable and allows the user to define the meaning of new formats in terms of existing formats as well as in terms of basic assembly instructions (whose meaning is built in).

It is unlikely that every machine user will want to write his own autocode: what is more likely is that he may wish to extend one of the standard languages to include statements suited to his own problem area."



#### A1.4 Summary

All of the extension mechanisms which we have discussed deal only with syntactic extensibility. Since data types are not context free they are best introduced by alternative methods. Some languages having syntactic extensibility also contain mechanisms for defining data types; eg ELF (Cheatham et al, 1968), PPL (Standish, 1969) and ECL (Wegbreit, 1971).

McIlroy suggests that macro facilities found in assembly languages be applied for extending high level languages. But, as Bowlden notes in his conclusions on McAlgol, a strategy using textual macros for preprocessing extended language programs denies meaningful error diagnostics.

Leavenworth's syntactic macro is conceptually simple, provides for limited conditional expansion and fits nicely into an LL(1) base language. Cheatham's syntactic macro is similar to Leavenworth's but is defined in the context of a simple precedence parse. With computational macros, one can express meanings which cannot be expressed in the base language and can produce more efficient code; but here one requires a knowledge of the implementation. The proposals of Schuman and Jorrand serve better as a model for examining the power of expression which can be attained with the syntactic macro than as a practical tool for language extension. The major contribution here is the use of predicates for imposing restrictions on macro calls which cannot be imposed by context free syntax; separating the predicate clause from the macro body is particularly important since the two clauses perform separate functions. Hammer's contribution is a more effective macro process. Distinguishing between those semantic actions which can be done at macro definition time and those which must be deferred to expansion time not only yields a more efficient macro process but also gives one a better understanding of compilation, the macro process and their synthesis. Excepting the detection of syntax



errors in the simple (type 1) bodies in Hammer's macros, none of the above proposals deal with the important problem of error diagnostics.

Our survey of syntactically extensible languages is by no means exhaustive. There are plenty of other examples; eg Irons' syntactic extensions to IMP (1970), Wegbreit's ECL (1971) and Standish's PPL (1969). An exhaustive survey of extensible languages is given by Solntseff and Yezerski (1974).

Algol C's extension mechanism is, on the surface, remarkably like the syntactic macro; but macro calls are recognised by means of a tree matching mechanism once the source program has been parsed. It is interesting to note the attempt to introduce new data types by way of syntactic extensions to the nonterminal <type>; yet the macro definitions required for specifying operations on new types are extremely complicated. The operator definition mechanisms in ELF and BALM illustrate a more successful strategy for this. New data objects are introduced by other means, eg by mode and functions, and a trimmed down version of Cheatham's syntactic macro is used for specifying the syntaxes for operator expressions on the new objects. Vidart's extensions to GSL are purely syntactic and illustrate an application of Leavenworth's syntactic macro to an existing programming language. By severely restricting the syntaxes of macro calls he is able to parse macro bodies at definition time and detect syntax errors where they are introduced; but Vidart allows neither alternative syntaxes in macro calls nor conditional expansion. Because ALEC is defined on top of a compiler compiler it can provide both formal (syntactic) macros for the naive user and informal macros whereby a more knowledgeable user may take advantage of the compiler compiler mechanisms in RCC. The restrictions imposed on the formal macro permit syntax error diagnostics in its definition and both kinds of macro provide a rich variety of parameter passing methods. Finally, the ECT system is a formal synthesis of compiler compiler techniques and syntactic extensibility

techniques for both extending and contracting languages. The rich variety of base languages offered by ECT relies, as with any compiler compiler based facility, on the use of an independent systems programming language.

## APPENDIX 2

### Summaries of $L_B$ and $L_E$

In the following sections we summarise the facilities in the illustrative base language  $L_B$  and the extension language  $L_E$ . The context free syntaxes for these languages are described by an extended BNF. Square brackets ([, ]) are used for enclosing optional symbol sequences, curly brackets ({, }) denote zero or more occurrences of a sequences and darkened parentheses ( (, ) ) bracket groups of alternatives.

#### A2.1 The base language $L_B$

##### A2.1.1 Syntax

We follow Turner and Morrison (1976) in describing the syntax of  $L_B$  using two sets of rules: a sequence of BNF rules and a sequence of type matching rules.

The context free syntax for  $L_B$  is as follows.

```

<program>      ::= <sequence> eof
<sequence>     ::= <declaration> ; <sequence>
                ::= <statement> {;<statement>}
<declaration> ::= let <identifier> = <expression>
                ::= structure <identifier> ((int|bool|str|ptr) <identifier>
                {, (int|bool|str|ptr) <identifier>})
                ::= decl <declaration> {;<declaration>} edec1
                ::= label <identifier>
                ::= declare <identifier> as <formal-parameter>
                ::= macro <target class> rule <macro template>
                [where <assertion>] means <macro body> endmacro
<statement>    ::= <var> := <expression>
                ::= if <expression> then <statement>
                ::= goto <name>
                ::= lab <name> {:<statement>}
                ::= writeln <expression>
                ::= writes <expression>
                ::= begin <sequence> end
<expression>   ::= <expr> [(=|>|<|>|<|=|is|isnt) <expr>]
<expr>         ::= [-] <term> {(+|-|or|concat) <term>}

```

```

<term>      ::= <factor> {(*|/|and) <factor>}
<factor>    ::= substr <expression>, <expression> of <factor>
             ::= begin <sequence> → <expression> end
             ::= ( <expression> )
             ::= ¬ <factor>
             ::= length <factor>
             ::= readi
             ::= reads ( <expression> )
             ::= <var>
             ::= <constant>
             ::= construct <name> (<expression> {,<expression>})
<var>      ::= <name> [( <var> )]
<name>     ::= <identifier>
<identifier> ::= } recognised by lexical analysis
<constant> ::= }

```

For specifying the type rules we use a notation proposed by Turner and Morrison (1976). A program is not well typed unless the type rules show it to be of type void. For example,

int + int ⇒ int

says that an expr formed from integer terms by the + operator is of type int and since no other rules for + are given, no other types of expr formed by it are legal. The meta-variables T, T<sub>1</sub>, T<sub>2</sub>, ..., T<sub>n</sub> stand for arbitrary well-typed sub-expressions; the same variable used consistently within one rule refers to the same type. For example,

begin void → T end ⇒ T

says that the type of the block expression as a whole is the type of the expression following the → operator. The type rules for L<sub>B</sub> follow.

```

void eof ⇒ void
  declaration ; void ⇒ void
void; void ⇒ void
T := T ⇒ void
if bool then void ⇒ void
goto lab ⇒ void
lab lab : void ⇒ void
writeln int ⇒ void
writes str ⇒ void
begin void end ⇒ void

```

$T (=| \neq) T \Rightarrow \text{bool}$   
 $\text{int} (<=|<|>|>=) \text{int} \Rightarrow \text{bool}$   
 $\text{int} (+|-|+|/) \text{int} \Rightarrow \text{int}$   
 $\text{bool} (\text{and}|\text{or}) \text{bool} \Rightarrow \text{bool}$   
 $\text{str } \underline{\text{concat}} \text{ str} \Rightarrow \text{str}$   
 $\text{ptr } (\underline{\text{is}}|\underline{\text{isnt}}) \text{ T-constructor} \Rightarrow \text{bool}$   
 $\underline{\text{substr}} \text{ int, int of str} \Rightarrow \text{str}$   
 $\underline{\text{begin}} \text{ void} \rightarrow \text{T } \underline{\text{end}} \Rightarrow \text{T}$   
 $(\text{T}) \Rightarrow \text{T}$   
 $\neg \text{bool} \Rightarrow \text{bool}$   
 $\underline{\text{length}} \text{ str} \Rightarrow \text{int}$   
 $\underline{\text{readi}} \Rightarrow \text{int}$   
 $\underline{\text{reads}} (\text{int}) \Rightarrow \text{str}$   
 $\underline{\text{construct}} (\text{T}_1, \dots, \text{T}_n)\text{-constructor} (\text{T}_1, \dots, \text{T}_n) \Rightarrow \text{ptr}$   
 $\text{T-field} (\text{ptr}) \Rightarrow \text{T}$

The following rules may be used for deducing the type of a name from its declaration; eg the first rule says <name> is associated with type T (Syntactically, a declaration makes an <identifier> a <name>).

1.  $\underline{\text{let}} \text{ <name> = T} \Rightarrow \text{<name> : T}$
2.  $\underline{\text{structure}} \text{ <name> } (\text{T}_1 \text{ <name}_1>, \dots, \text{T}_n \text{ <name}_n>) \Rightarrow$   
 $\text{<name> : } (\text{T}_1, \dots, \text{T}_n)\text{-constructor}$   
 $\text{<name}_1>: \text{T}_1\text{-field}$   
 $\vdots$   
 $\text{<name}_n>: \text{T}_n\text{-field}$
3.  $\underline{\text{label}} \text{ <name>} \Rightarrow \text{<name> : lab}$
4.  $\underline{\text{declare}} \text{ <name> as T} \Rightarrow \text{<name> : T}$

The names introduced by let and declare are variables whereas the names introduced by structure and label are not and can neither be assigned, receive assignments nor can they be compared for equality. The scope of a name is that <sequence> in which it was declared.

An  $L_B$  program must be both well-formed according to the extended BNF rules and well-typed according to the type rules.

### A2.1.2 Explanation of various constructions

Since the meaning of most  $L_B$  constructs can be gleaned from the BNF and type rules we review only the unusual ones here.

1. Constants may be denoted for each of the four simple types:
  - (a) int - an integer constant consisting of a sequence of decimal digits;
  - (b) bool - a boolean constant true or false;
  - (c) str - a string of zero or more characters which are enclosed with double quotes; eg "" is the empty string and """" is a string containing one double quote (two double quotes represent one in a string); and
  - (d) ptr - the special null pointer.
2. Labels may only be used in conjunction with goto's. Labels must be declared and branches may not be made into a begin ... end block. Labels and goto's used in a macro body are local to that body. Anyone who does not like goto's may delete the corresponding BNF rules.
3. String manipulation facilities include a length operator, an infix concatenation operator, concat, and a substring operator, substr. Eg if *i* and *j* are integers, and *r* and *s* are strings, then all of the following  $L_B$  expressions are true.
  - (a) "top" concat "cat" = "topcat"
  - (b) substr 2,5 of "outside" = "utsid"
  - (c) length substr *i,j* of *s* = *j*
  - (d) length (*s* concat *r*) = length *s* + length *r*
  - (e) (*i* <=length *s*) or (substr 1, *i* of *s* = "")

Ie the empty string is returned for undefined substrings.
4. Structured types may be declared with structure ...; objects of these types may be constructed using construct which returns a pointer to the object constructed (see the type rules in A2.2.1). Since the types of objects pointed to (by values of type ptr)

cannot be determined at run time,  $L_B$  has the run time operators is and isnt for querying the types of structured objects.

5. Generic declarations of the form

declare <identifier> as <formal parameter ref>

may be used for declaring names only in macro bodies. As implied by the type rules, the name takes on the type of the call time actual parameter corresponding to the formal parameter reference. Any formal parameter used in a generic declaration must be tagged with eval in the template (thus causing actual parameters to be passed by reference).

6. Macro declarations (macro ...) are like macro definitions, described in A2.2, but are local to the begin ... end block in which they are declared. If a local  $L_P$  form conflicts with the surrounding grammar, the local form takes precedence in any parsing decision. A macro declaration takes effect as soon as it is scanned in the  $L_P$  source text and remains in effect until the end of the block in which it was declared (so it should not be nested in another macro definition nor call). The macro body may contain free names which have been declared globally to the macro declaration; if a type 1 macro body is tagged with eval then all free names are bound at macro definition time.

7. Input and output

- (a) writeln n, causes the integer expression n to be written out.
- (b) writes s, writes out the string expression s.
- (c) readi returns as its value the next integer from input.
- (d) reads (n) returns as its value the next n characters from input.

Eg one might introduce readch with the following simple macro

macro \$factor rule 'readch' means [reads (1)] endmacro

A2.2 The extension language  $L_E$

A2.2.1 An  $L_E$  program

$\langle L_E \text{ program} \rangle ::= \langle L_E \text{ definition file} \rangle \text{ program } \langle L_P \text{ program} \rangle$   
 $\langle L_E \text{ definition file} \rangle ::= \{ \langle \text{definition} \rangle ; | \langle \text{replacement} \rangle ; \}$   
 $\{ \langle \text{deletion} \rangle ; \}$

An  $L_E$  program consists of a (possibly empty) file of definitions for extending (and contracting)  $L_B$  to  $L_P$  and a user program written in  $L_P$ .  $L_P$  (initially equivalent to  $L_B$ ) is incrementally modified by successive macro definitions, replacements and deletions in the file. Each modification must preserve the LL(1) condition imposed on  $L_B$ ; this is relatively easy to check since no  $L_P$  BNF rule may produce the empty string.

A2.2.2 A macro definition

$\langle \text{definition} \rangle ::= \text{define } \langle \text{target class} \rangle$   
 $\text{ rule } \langle \text{macro template} \rangle$   
 $[\text{where } \langle \text{assertion} \rangle]$   
 $\text{ means } \langle \text{macro body} \rangle \text{ endif }$   
 $\langle \text{target class} \rangle ::= \langle \text{class name} \rangle$

A macro definition introduces a new BNF rule for the target class. Its syntax is specified in the macro template and its meaning is given by the macro body. If an assertion is present then it must be satisfied by any macro call matching the template. The macro process is described in section 3.5. A macro definition applies to subsequent  $L_E$  definitions and the  $L_P$  program.



### A2.2.2.1 The macro template

```

<macro template> ::= <template>
<template> ::= {<basic>|<formal parameter>|<sub-template>}
<basic> ::= (a basic token symbol enclosed in single quotes)
<formal parameter> ::= <class name> [. ( type | eval) ]
<sub-template> ::= [<identifier>:] ( (? <template> ?) |
                                   (* <template> *) |
                                   ( <template> { |<template> } ) )

<type> ::= <simple type> [field]
          ::= (<simple type> {,<simple type>})
<simple type> ::= int|bool|str|ptr

```

A macro template specifies the syntax of macro calls in terms of a template which, in turn, is a sequence of basic tokens, formal parameters and/or subtemplates; no template may represent the empty string.

A formal parameter is represented by the (possibly subscripted) name of any of the twelve syntactic classes (nonterminals) in  $L_B$ 's grammar; actual parameters in a macro call are parsed against these corresponding syntactic classes. If a formal parameter is tagged with either a type denotation (to which the type of an actual parameter must correspond) or eval then corresponding actual parameters will be type checked at call time and passed by reference; otherwise, actual parameters are passed textually. The type denotations (ie <type>) correspond to the notation used in the type rules for  $L_B$  (section A2.1.1); (...) means (...) - constructor.

Sub-templates are used for specifying alternative syntaxes for calls. Each of the three kinds of sub-template is described below; subscripted  $t$ 's stand for arbitrary nested templates.

1. The optional sub-template (?  $t_1$  ?) represents an optional occurrence of  $t_1$ ; ie  $t_1$  may or not be matched in a call.
2. The list sub-template (\*  $t_1$  \*) represents a list of zero or more occurrences of  $t_1$ ; ie  $t_1$  may be matched zero or more times.

3. The alternative sub-template  $(t_1|t_2|\dots|t_n)$  represents a collection of alternative nested templates, one of which must be matched in a call.

Any grammatical ambiguities introduced by sub-templates are resolved by the following rules (assuming a top down left-to-right parse).

1. In an optional subtemplate,  $t_1$  is always matched where possible.
2. In a list subtemplate,  $t_1$  is successively matched wherever possible.
3. In an alternative subtemplate, the first possible  $t_i$  (from left-to-right) is always matched.

Any sub-template may be given a name by which it may be referred to in the assertion or macro body.

#### A2.2.2.2 The assertion

```

<assertion> ::= <predicate>
              ::= list <assertion> {,<assertion>} end
              ::= given <subs> then <assertion> else <assertion>
              ::= forall <subs> : <assertion>
              ::= choosing <sub-name> from
                  list <assertion> {,<assertion>} end
              ::= if <predicate> then <assertion> else <assertion>
              ::= assert <assertion> : <string constant>
<subs>      ::= <sub-name> {,<sub-name>}
<sub-name>  ::= <identifier> [.<sub-name>]
    
```

Assertions impose certain context sensitive restrictions on macro calls which cannot be imposed by context free methods. An assertion is either satisfied or unsatisfied; an assertion in the where clause of a macro definition must be satisfied by every macro call. The rules by which assertions are satisfied are given below; subscripted A's stand for arbitrary assertions and subscripted S's stand for the names of sub-templates appearing in the macro template.

1. If the assertion is simply a predicate then it is satisfied iff (if and only if) the predicate is true.

2. list  $A_1, \dots, A_n$  end is satisfied iff each of  $A_1, \dots, A_n$  is satisfied.
3. given  $S_1, \dots, S_n$  then  $A_1$  else  $A_2$  is satisfied
  - (1) if all optional sub-templates  $S_1, \dots, S_n$  were matched in the call, then iff  $A_1$  is satisfied, and
  - (2) iff  $A_2$  is satisfied otherwise.
4. forall  $S_1, \dots, S_n$  :  $A_1$  is satisfied iff  $A_1$  is satisfied for each successive match of the list sub-templates named  $S_1, \dots, S_n$ .  
 Ie  $A_1$  must be satisfied for the first matches of  $S_1, \dots, S_n$ , for the second matches of  $S_1, \dots, S_n$  and so on. Note that this means that call of  $S_1, \dots, S_n$  must be matched an equal number of times.
5. choosing  $S_1$  from list  $A_1, \dots, A_n$  end is satisfied iff
  - (1) the first alternative of the alternative sub-template  $S_1$  was matched in the call and  $A_1$  is satisfied, or
  - (2) the second alternative of  $S_1$  was matched and  $A_2$  is satisfied, or ...
  - ⋮
  - (n) the  $n^{\text{th}}$  alternative of  $S_1$  was matched and  $A_n$  is satisfied.
6. Assuming  $p$  is a predicate, if  $p$  then  $A_1$  else  $A_2$  is satisfied
  - (1) if  $p$  is true then iff  $A_1$  is satisfied, and
  - (2) iff  $A_2$  is satisfied, otherwise.
7. assert  $A_1$  : <string constant> is satisfied iff  $A_1$  is satisfied; as a side effect, the string constant is written out into the source text (following the call) iff  $A_1$  was not satisfied.

Sub-template names used in the assertion (or the macro body) correspond to the names given to sub-templates in the macro template. For deeply nested sub-templates, a path name must be given; eg

$S_1.S_2.S_3$

refers to the sub-template named  $S_1$  which is nested in the sub-template named  $S_2$  which is nested in the sub-template named  $S_3$ .

The names of formal parameters and sub-templates in the macro template are scoped within corresponding  $L_E$  constructs in the assertion (and macro body) for prohibiting references to parameters and sub-templates which are not matched at call time. The scope rules for the three  $L_E$  constructs corresponding to sub-templates are given below; subscripted  $X$ 's stand for arbitrary assertions (or for arbitrary text segments in the macro body) and subscripted  $S$ 's stand for sub-template names.

1. In given  $S_1, \dots, S_n$  then  $X_1$  else  $X_2$ , formal parameters and sub-templates nested in the optional sub-templates  $S_1, \dots, S_2$  may be named in  $X_1$  but not in  $X_2$ .
2. In forall  $S_1, \dots, S_n : X_1$ , names in  $X_1$  may refer only to formal parameters and sub-templates which are nested in the list sub-templates  $S_1, \dots, S_n$ .
3. In choosing  $S_1$  from list  $X_1, \dots, X_n$  end, names in  $X_i$  may refer only to formal parameters and sub-templates which are nested in the  $i^{\text{th}}$  alternative of the alternative sub-template  $S_1$ , for  $i = 1, \dots, n$ .

#### A2.2.2.3 The macro body

```

<macro body> ::= [eval :] [<Lp source segment>]
               ::= <text segment>

<text segment> ::= [<class name> :] [<Lp source segment>]
                 ::= list <text segment> {,<text segment>} end
                 ::= given <subs> then <text segment> else <text segment>
                 ::= forall <subs> : <text segment>
                 ::= choosing <sub-name> from
                    list <text segment> {,<text segment>} end
                 ::= if <predicate> then <text segment> else <text segment>

```

The macro body specifies the (current  $L_p$ ) text which is to replace a macro call; actual parameters to the call are substituted for corresponding formal parameter references in the body.

If the macro body is simply a bracketed string of  $L_p$  source text (possibly tagged by eval) then it is a type 1 macro body. A type 1 macro body may be parsed (against the target class) at macro definition

time for diagnosing errors. If all formal parameters in the template have been tagged with type denotations (if they represent objects having non-void types) and eval (for void types) then the body may be diagnosed for type violations; in this case, all names in the body are bound at definition time.

Otherwise, the macro body is a text segment represented by a (possibly compound)  $L_E$  construct. The actual  $L_P$  source segment represented by each construct is given below; subscripted S's stand for names of sub-templates in the macro template and subscripted T's stand for arbitrary (nested) text segments.

1. If the text segment is simply a bracketed string of  $L_P$  source text then its value is precisely that string of  $L_P$  text. If the bracketed string has been tagged with a syntactic class name then the string may be parsed against the corresponding syntactic class for detecting syntax errors at macro definition time.
2. list  $T_1, \dots, T_n$  end returns as its value the string of  $L_P$  source text formed by the concatenation of  $T_1, \dots$ , and  $T_n$ .
3. given  $S_1, \dots, S_n$  then  $T_1$  else  $T_2$  returns as its value the text for  $T_1$  iff each optional sub-template  $S_1, \dots, S_n$  was matched in the call and the text for  $T_2$  otherwise.
4. forall  $S_1, \dots, S_n : T_1$  where each of the list sub-templates  $S_1, \dots, S_n$  was matched an equal number of times (say  $m$  times), returns a concatenation of  $m$  text segments:
  - (1) the segment  $T_1$  corresponding to the first match of  $S_1, \dots, S_n$ ;
  - (2) the segment  $T_1$  corresponding to the second match;
  - $\vdots$
  - ( $m$ ) the segment  $T_1$  corresponding to the  $m^{\text{th}}$  match.
5. choosing  $S_1$  from list  $T_1, \dots, T_n$  end returns as its value the text for  $T_i$  where the  $i^{\text{th}}$  alternative of the alternative sub-template  $S_1$  was matched in the call.
6. if  $p$  then  $T_1$  else  $T_2$ , where  $p$  is a predicate, returns as its value

the text for  $T_1$  if  $p$  is true and  $T_2$  otherwise.

The names of nested formal parameters and sub-templates referred to in the  $T_i$ 's above must conform to the same scope rules imposed on assertions in order to prevent references to parameters and sub-templates not matched in a call.

#### A2.2.2.4 Predicates and formal parameter references

```

<predicate> ::= match (<a-type>, <a-type>)
              ::= eqlen (<list length>, <list length>)
              ::= not (<predicate>)
              ::= or (<predicate>, <predicate>)
              ::= and (<predicate>, <predicate>)
              ::= true | false

<a-type> ::= type <formal parameter ref>
           ::= <type>

<list length> ::= <sub-name> | <integer constant>

<formal parameter ref> := <class name> [.<sub-name>]

```

Predicates are used in assertions and in the macro body (for conditional text replacement). The predicate match ( $a_1, a_2$ ) is true if the types  $a_1$  and  $a_2$  are equivalent and false otherwise. Each of  $a_1$  and  $a_2$  may be either a constant type denotation or an application of type to a formal parameter reference for ascertaining the type of a corresponding actual parameter. In the latter instance, the formal parameter must be tagged with eval in the macro template. The predicate eqlen ( $S_1, S_2$ ) is true iff the list sub-templates  $S_1$  and  $S_2$  were matched an equal number of times in the call; note that either of  $S_1$  and  $S_2$  may be replaced by an integer constant for checking the length of the other. The values returned by the remaining compound and constant predicates are obvious.

A formal parameter reference is used in the macro body (or an assertion) for referring to actual parameters matching a corresponding formal parameter in the template. If the formal parameter is nested within a sub-template then the full path name must be given.

### A2.2.3 Replacements and deletions

```

<replacement> ::= replace <target class>
                  rule <abbreviated template> [...]
                  by <macro template>
                  [where <assertion>]
                  means <macro body> endef

<deletion> ::= delete <target class> rule <abbreviated template> endef

```

A macro replacement is like a macro definition but the new  $L_P$  form, described by the macro template, replaces an old  $L_P$  form, described by the abbreviated template, for subsequent definitions and the  $L_P$  program. The abbreviated template may be either the entire template for the old form or a prefix of it; in any case it must uniquely identify the form which is to be replaced and the set of possible starting tokens for the new form must be equivalent to that for the old form. The macro body may make use of the old form for defining the meaning of the new form.

A macro deletion deletes that  $L_P$  form for the target class which is identified by the abbreviated template. Again, the abbreviated template must identify a unique  $L_P$  form.

Neither replacements nor deletions have any effect on the text in the macro bodies of preceding definitions in the  $L_E$  definition file.

### A2.3 Lexical representation - the micro syntax

Those syntactic entities for which we have not given precise representations are part of the micro syntax and depend on such things as the character set available to a particular implementation. The micro syntax which guides the lexical analyser in our implementation is as follows.

1. <constant>. Constant values are represented in the way they are described for  $L_B$  in section A2.1.2.



2. <identifier>. Identifiers are represented by lowercase letters where they are available and by the @ symbol followed by uppercase letters elsewhere; eg i, @I, variable, @VARIABLE.
3. <class name>. A syntactic class name is represented by the \$ followed by its name in uppercase letters; subscripts are represented by single digits appended to the end of the name. eg \$PROGRAM, \$STATEMENT1.
4. Reserved words are represented by uppercase words, eg BEGIN for begin and GIVEN for given. Operators are represented by concatenations of punctuation symbols; eg (? , \*), +. Two adjacent operators must be separated by a blank. On cards, the  $L_p$  text brackets [ and ] are represented by (. and .) respectively.

#### A2.4 Two additional examples

Examples of macro definitions, replacements and deletions appear throughout Chapters 2 and 3. We give two additional examples which involve string operations.

Firstly, the following simple replacement may be used for redefining the substr operator (which now returns a substring of a given length beginning at a given character position) so that it returns a substring from one character position to another.

```

replace $factor
  rule 'substr' ...
    by 'substr' $expression1.int 'to' $expression2.int
      'of' $expression3.str
  means
    eval : [substr $expression1, ($expression2) - ($expression1) + 1
      of $expression3]
  endef

```

Secondly, the following definition introduces a new interleave form for returning the concatenation of two lists of string expressions once they have been interleaved; the lists must be of equal length.



```
define $factor
  rule 'interleave' $expression1.str S1 : (* ',' $expression.str *)
          'with' $expression2.str S2 : (* ',' $expression.str *)
  where
    assert eqlen (S1,S2) : "string lists of unequal length"
  means
    list
      [( ($expression1) concat ($expression2)),
        forall S1,S2 :
          [concat ($expression.S1) concat ($expression.S2)],
        [] ]
    end
  endef
```

For example

```
writes interleave "Top","Under","Big" with "cat, ","dog ","bird";
```

would, in an  $L_p$  program, cause the following string to be output.

```
"Topcat, Underdog, Bigbird"
```

### APPENDIX 3

#### The Object Machine

$L_p$  programs are compiled to code for an object machine which has a stack, on top of which simple variables are allocated space and expressions are evaluated, and a heap on which strings and structured objects are stored. Each type of simple variable requires the same amount of storage space (1 cell) and, since  $L_p$  has no procedures, stack addresses for variables may be determined at compile time. The machine was not designed as a general purpose computer but only to illustrate the code generation phase of our compiler for  $L_p$ .

Each structure declaration in  $L_p$  associates a unique integer "trademark", borrowed from Morrison (1978), with the declared T-constructor name and also associates consecutive trademarks for successive field names of the T-constructor. For example, the structure declaration

structure binary (int info, ptr leftbranch, ptr rightbranch)

might associate the trademark 6 to binary, 7 to info, 8 to leftbranch and 9 to rightbranch; a subsequent T-constructor name would be assigned 10 as a trademark. Each structured object on the heap is prefixed by both a trademark denoting its type and a field count. These values are used for validating the application of field names to (pointers to) structured objects, computing offsets and for implementing the operators is and isnt. String values are represented on the stack by pointers to special string structures on the heap; each string structure on the heap is prefixed with a trademark denoting it as a string and a character count. Trademarks also provide information necessary to garbage collection.

Labels are explicitly denoted by the pseudo-instruction LAB which associates a unique integer with its position in the object deck; all branch instructions refer to these integer labels. These might be

replaced with their actual instruction word addresses in a first pass of a loader to the interpreter.

The forty-one machine instructions are described below. Addresses, labels and trademarks are represented by integers. The constant null pointer is represented by 0 and the boolean constants true and false are represented by 1 and 0 respectively.

1. Instructions for accessing and moving data.
  - (a) LOADA address - Load address onto the stack.
  - (b) LOADPA address - Load pointer's address onto the stack.
  - (c) LOADC constant - Load constant onto the stack.
  - (d) STORE - Pop the top value off the stack and store it at the address popped off the stack.
  - (e) CONTENT - Replace the address on the top of the stack with the content at that address.
  - (f) LOADTM trademark - Load the trademark onto the stack.
  - (g) OFFSET trademark - Apply trademark (for a field name) to the pointer (to a structured object) on the stack for replacing that pointer on the stack with the address of the field named.
  - (h) STORETOP - Pop the top value off the stack and save it in the (only) register.
  - (i) RESETSP address - Reset the stack pointer to a new (top) address.
  - (j) LOADTOP - Load the value in the (only) register onto the stack.
  - (k) CONSTRUCT trademark, n - Construct a new structured object (on the heap) of type trademark and with n fields consecutively defined by n values popped off the top of the stack; the first value popped off defines the n<sup>th</sup> field. Load a pointer to this object onto the stack.

2. Branch instructions

- (a) LAB label - A pseudo-instruction for implanting a label into code.
- (b) GOTO label - Branch unconditionally to label.
- (c) JUMPF label - Pop the top (boolean) value off the stack; if false then branch to label.
- (d) JUMPTF label - If top (boolean) value is true, branch to label.
- (e) JUMPTFF label - If top (boolean) value is false, branch to label.
- (f) HALT - Halt the process.

3. Relational operations

- (a) EQ, NE, LT, GT, LE, GE - Pop two values of the stack, firstly X and then Y. If Y <relation> X holds then load true onto the stack; otherwise load false onto the stack.
- (b) CMPSTR - This instruction is always followed by one of the instructions in (a). Two string pointers are popped off the stack. The strings pointed to are compared character by character (a shorter string is padded on the right with blanks) for testing the relation indicated in the next instruction. If the relations holds, true is loaded onto the stack; otherwise false is loaded.

In the next two instructions, a trademark and then a pointer are popped off the stack. The trademark is compared for equality with that for the object (first field) pointed to by the pointer.

- (c) IS - If the two trademarks are equivalent then load true onto the stack; otherwise load false onto the stack.
- (d) ISNT - If the two trademarks are not equivalent then load true onto the stack; otherwise load false onto the stack.

4. String operations

- (a) SUBSTR - Pop a pointer p, an integer (length) n, and an integer (character position) i off the stack. Construct a new string (on the heap) of length n, copied from character i in the string pointed to by p. Load a pointer to the string onto the stack.
- (b) CONCAT - Pop a pointer p and a pointer q off the stack. Construct a new string on the heap from a copy of the string pointed to by p concatenated onto the end of a copy of that pointed to by q. Load a pointer to the new string onto the stack.
- (c) LEN - Pop a pointer to a string off the stack and load the integer length of that string (in second field) onto the stack.

5. Arithmetic and boolean operations

For each of the operations (a) and (b), pop one value X off the stack; for each of (c) - (h), pop two values, X and then Y, off the stack.

- (a) NEGATE - Load the two's complement of X onto the stack.
- (b) NOT - Load the one's complement of X onto the stack.
- (c) ADD - Load  $Y + X$  onto the stack.
- (d) SUB - Load  $Y - X$  onto the stack.
- (e) MULT - Load  $Y * X$  onto the stack.
- (f) DIV - Load  $Y / X$  onto the stack.
- (g) AND - Load Y and X onto the stack.
- (h) OR - Load Y or X onto the stack.

6. Input and output operations

- (a) READI - Load the next integer read from input onto the stack.
- (b) READS - Pop an integer value n off the stack. Construct a new string (on the heap) from the next n characters read from

the input file. Load a pointer to the new string onto the stack.

- (c) WRITEI - Pop an integer off the stack and write it to the output file.
- (d) WRITES - Pop a pointer to a string off the stack and write that string to the output file.

REFERENCES

1. Aho, A.V. and Ullman, J.D. (1972). The Theory of Parsing, Translation and Compiling : Vol. 1, Prentice-Hall, Englewood Cliffs, N.J. pp. 333 - 361.
2. Aho, A.V. and Ullman, J.D. (1973). The Theory of Parsing, Translation and Compiling : Vol. 2, Prentice-Hall, Englewood Cliffs, N.J. pp. 627 - 631.
3. Ammann, U. (1973). The method of structured programming applied to the development of a compiler. Proc. International Symposium on Computing (1973), ed. Gunther et al, North Holland, Amsterdam (1974), pp. 93 - 99.
4. Berthaud, M., Clauzel, D. and Jacolin, M. (1972). Grenoble Systems Language. Etude no. FF2.0133, IBM Centre Scientifique de Grenoble.
5. Bowlden, H.J. (1971). Macros in higher-level languages. Proc. of the International Symposium on Extensible Languages, ed. Schuman, S.A., SIGPLAN Notices, Vol. 6, No. 12, pp. 39 - 44.
6. Brooker, R.A. and Morris, D. (1962). A general translation program for phrase structure languages. Journal of the ACM, Vol. 9, pp. 39 - 44.
7. Brown, P.J. (1967). The ML/I macro processor. Comm. of the ACM, Vol. 10, pp. 618 - 623.
8. Brown, P.J. (1969). A survey of macro processors. Annual Review in Automatic Programming, Vol. 6, Pergamon Press, London, pp. 37 - 88.
9. Brown, P.J. (1974). Macro Processors and Techniques for Portable Software, John Wiley and Sons, London.
10. Campbell, W.R. (1978). A compiler definition facility based on the syntactic macro. Computer Journal, Vol. 21, No. 1, pp. 35 - 41.
11. Cheatham, T.E. Jr. (1966). The introduction of definitional facilities into higher level programming languages. AFIPS Proceedings (FJCC), Vol. 33, Part 2, pp. 937 - 984.
12. Cheatham, T.E. Jr., Fischer, A. and Jorrand, P. (1968). On the basis for ELF - an extensible language facility. AFIPS Proceedings (FJCC), Vol. 33, Part 2, pp. 937 - 984.
13. Cole, A.J. (1976). Macro Processors, Cambridge University Press, Cambridge.
14. Earley, J. (1974). High level operations in automatic programming. Proc. of a Symposium on Very High Level Languages, SIGPLAN Notices, Vol. 9, No. 4, pp. 34 - 42.
15. Galler, B.A. and Perlis, A.J. (1967). A proposal for definitions in Algol. Comm. of the ACM, Vol. 10, pp. 204 - 219.
16. Griffiths, M. and Peltier, M. (1968). Grammar transformation as an aid to compiler production. Etude no. FF2.0057, IBM Centre Scientifique de Grenoble.



17. Hammer, M. (1971). An alternative approach to macro processing. Proc. of the International Symposium on Extensible Languages, ed. Schuman, S.A., SIGPLAN Notices, Vol. 6, No. 12, pp. 58 - 64.
18. Harrison, M.C. (1970). BALM - an extendable list-processing language. AFIPS Proceedings (SJCC), Vol. 36, pp. 507 - 511.
19. Irons, E.T. (1970). Experience with an extensible language. Comm. of the ACM, Vol. 13, pp. 31 - 40.
20. Johnson, S.C. (1975). YACC - yet another compiler-compiler. UNIX Documents, Bell Laboratories, Murray Hill, N.J.
21. Jorrand, P (1976). Private communication, 12 October 1976, St. Andrews.
22. Knuth, D.E. (1971). Top down syntax analysis. Acta Informatica, Vol. 1, pp. 79 - 110.
23. Leavenworth, B.M. (1966). Syntax macros and extended translation. Comm. of the ACM, Vol. 9, pp. 790 - 793.
24. McIlroy, M.D. (1960). Macro instruction extensions of compiler languages. Comm. of the ACM, Vol. 3, pp. 214 - 220.
25. McKeeman, W.M., Horning, J.J. and Wortman, D.B. (1970). A Compiler Generator, Prentice-Hall, Englewood Cliffs, N.J.
26. Mooers, C.N. (1966). TRAC, a procedure-describing language for the reactive typewriter. Comm. of the ACM, Vol. 9, pp. 215 - 219.
27. Morrison, R. (1978). Algol R. Internal report, Department of Computational Science, University of St. Andrews.
28. Napper, R.B.E. (1973). RCC reference manual. Department of Computer Science, University of Manchester.
29. Napper, R.B.E. and Fisher, R.N. (1976). ALEC - a user extensible scientific programming language. Computer Journal, Vol. 19, pp. 25 - 31.
30. Ollongren, A. (1974). Definition of Programming Languages by Interpreting Automata. Academic Press, London, pp. 149 - 170.
31. Richards, M. (1969). BCPL : a tool for compiler writing and system programming. AFIPS Proceedings (SJCC), Vol. 34, pp. 557, 556.
32. Ritchie, D.M. (1973). C reference manual. Bell Laboratories, Murray Hill, N.J.
33. Schuman, S.A. and Jorrand, P.A. (1970). Definition mechanisms in extensible programming languages. IBM Centre Scientifique de Grenoble.
34. Solntseff, N. and Yezerski, A. (1972). ECT - an extensible contractible translator system. Information Processing Letters, Vol. 1, pp. 97 - 99.
35. Solntseff, N. and Yezerski, A. (1974). A survey of extensible programming languages. Annual Review in Automatic Programming, Vol. 7 (part 5), pp. 267 - 307.

37. Standish, T.A. (1975). Extensibility in programming language design. AFIPS Proceedings (NCC), Vol. 44, pp. 287 - 290.
38. Turner, D. and Morrison, R. (1976). Towards more portable compilers. Internal report, TR/76/5, Department of Computational Science, University of St. Andrews.
39. Turner, D.A. (1977). Error diagnosis and recovery in one pass compilers. Information Processing Letters, Vol. 6, pp. 113 - 115.
40. Vidart, J. (1974). Extensions syntaxiques dans une context LL(1). Thèse pour obtenir le grade de Docteur do troisième cycle, University of Grenoble.
41. Warshall, S. (1962). A theorem on boolean matrices. Journal of the ACM, Vol. 9, pp. 11 - 12.
42. Wegbreit, B. (1971). The ECL programming system. AFIPS Proceedings (FJCC), Vol. 39, pp. 253 - 262.
43. Wirth, N. and Hoare, C.A.R. (1966). A contribution to the development of Algol. Comm. of the ACM, Vol. 9, pp. 413 - 432.
44. Wirth, N. and Weber, H. (1966). EULER : a generalization of ALGOL and its formal definition (Parts 1 and 2), Comm. of the ACM, Vol. 9, pp. 13 - 23, 89 - 94.
45. Wirth, N. (1968). PL360, a programming language for the 360 computers. Journal of the ACM, Vol. 15, pp. 37 - 74.
46. Wirth, N. (1971). The programming language Pascal. Acta Informatica, Vol. 1, pp. 35 - 63.
47. Wirth, N. (1977). What can we do about the unnecessary diversity of notation for syntactic definitions? Comm. of the ACM, Vol. 20, pp. 822 - 823.
48. van Wijngaarden, A. et al (1975). Revised report on the algorithmic language Algol 68. Acta Informatica, Vol. 5.

## EXAMPLE LISTINGS

The following pages contain listings for illustrating some of the definitional facilities and error diagnostics in  $L_E$ . The nine examples are briefly described below.

1. Definitions for introducing a statement for summing a list of integers and for introducing a simple variable declaration.
2. Macro declarations for introducing integer stack operations.
3. Definitions for introducing a while statement, for statement (with an implicitly declared variable), and a factorial operator.
4. Diagnosis of a syntax error in a macro body.
5. Diagnosis of a type violation in a macro body.
6. Diagnosis of a type violation, having to do with structure names, by way of a typed formal parameter in a macro template.
7. Diagnosis of a type violation found by way of an assertion.
8. Diagnosis of a syntax error caused by the premature deletion of the if statement. Also, extension of the if statement and the introduction of an if-then-else expression and a case statement.
9. Illustration of the object code produced by the compiler.

Example 1

```

DEFINE &EXPRESSION
  RULE 'SUM' @OPT: (? 'SQUARES' 'OF' ?) &EXPRESSION1
    @OTHERS: (* ( ' ' ] '&' ) ~&EXPRESSION2 *)
MEANS
  LIST
    GIVEN @OPT
      THEN ( ( &EXPRESSION1 ) * ( &EXPRESSION1 ) . )
      ELSE ( ( &EXPRESSION1 ) . ) ,
    FORALL @OTHERS:
      GIVEN @OPT
        THEN ( . + ( &EXPRESSION2.@OTHERS ) *
                  ( &EXPRESSION2.@OTHERS ) . )
        ELSE ( . + ( &EXPRESSION2.@OTHERS ) . )
      END
    ENDEF;
ENDEF;

DEFINE &DECLARATION
  RULE @TYPE: ( 'INT' ] 'BOOL' ] 'STR' ] 'PTR' ) &ID
MEANS
  CHOOSING @TYPE FROM
  LIST
    &DECLARATION: ( . LET &ID = 0 . ) ,
    &DECLARATION: ( . LET &ID = FALSE . ) ,
    &DECLARATION: ( . LET &ID = "" . ) ,
    &DECLARATION: ( . LET &ID = NULL . )
  END
ENDEF;

PROGRAM
BEGIN
  INT @X; INT @Y; INT @Z;
  @X := 5;
  @Y := READI;
  @Z := SUM SQUARES OF @X+1 & SUM @X-1, @Y & @Y+1;
  WRITEI SUM SQUARES OF @Z & ( SUM @X & @Y ) & @X
END
EDF

```

000.26 SECONDS IN EXECUTION  
#EXECUTION TERMINATED  
#

Example 2

PROGRAM

BEGIN

```
STRUCTURE @NEW( INT @INFO, PTR @NEXT ) ;  
LET @I = 10;  
LET @J = 20;  
LET @TOP = NULL;
```

MACRO &STATEMENT

RULE 'PUSH' &EXPRESSION.INT

MEANS ~

```
EVAL: ( . @TOP := CONSTRUCT @NEW( &EXPRESSION, @TOP ) . )  
ENDMACRO;
```

MACRO &FACTOR

RULE 'POP'

MEANS ~

```
EVAL: ( . BEGIN  
                LET @RESULT = 0;  
                IF @TOP != NULL THEN  
                BEGIN  
                    @RESULT := @INFO( @TOP ) ;  
                    @TOP := @NEXT( @TOP )  
                END  
                -> @RESULT  
            END . )
```

ENDMACRO;

```
WRITEI @I + @J;  
PUSH @I + @J;  
PUSH @J + 1;  
PUSH POP * POP;  
WRITEI POP
```

END

EOF

000.23 SECONDS IN EXECUTION

#EXECUTION TERMINATED

#

Example 3

```
DEFINE %STATEMENT
  RULE 'WHILE' %EXPRESSION.BOOL 'DO' %STATEMENT.EVAL
  MEANS
```

```
  EVAL:
```

```
    (. BEGIN
```

```
      LABEL @LOOP;
```

```
      LAB @LOOP: IF %EXPRESSION THEN
```

```
        BEGIN
```

```
          %STATEMENT;
```

```
          GOTO @LOOP
```

```
        END
```

```
      END .)
```

```
  ENDEF;
```

```
DEFINE %STATEMENT
```

```
  RULE 'FOR' %ID ':' %EXPRESSION1.INT 'TO' %EXPRESSION2.INT
```

```
  'DO' %STATEMENT
```

```
  MEANS
```

```
    (. BEGIN
```

```
      LET %ID = %EXPRESSION1;
```

```
      WHILE %ID <= ( %EXPRESSION2 ) DO
```

```
        BEGIN
```

```
          %STATEMENT;
```

```
          %ID := %ID + 1
```

```
        END
```

```
      END .)
```

```
  ENDEF;
```

```
DEFINE %FACTOR
```

```
  RULE 'FACTORIAL' %FACTOR.INT
```

```
  MEANS
```

```
    EVAL:
```

```
      (. BEGIN
```

```
        LET @FACT = 1;
```

```
        FOR @I := 2 TO %FACTOR DO @FACT := @FACT * @I
```

```
        -> @FACT
```

```
      END .)
```

```
  ENDEF;
```

```
PROGRAM
```

```
  BEGIN
```

```
    LET @LIMIT = READI;
```

```
    FOR @I := 1 TO @LIMIT DO WRITEI FACTORIAL @I
```

```
  END
```

```
  EOF
```

```
000.24 SECONDS IN EXECUTION
```

```
#EXECUTION TERMINATED
```

```
#
```

Example 4

```

DEFINE %STATEMENT
  RULE 'WHILE' %EXPRESSION.BOOL 'DO' %STATEMENT.EVAL
  MEANS
    EVAL:
      (. BEGIN
        LABEL @LOOP;
        LAB @LOOP: IF %EXPRESSION THEN
          BEGIN
            %STATEMENT;
            GOTO @LOOP
          END
        END .)
  ENDEF;

```

```

DEFINE %STATEMENT
  RULE 'FOR' %ID ':' %EXPRESSION1.INT 'TO' %EXPRESSION2.INT
    'DO' %STATEMENT
  MEANS
    (. BEGIN
      LET %ID = %EXPRESSION1;
      WHILE %ID <= ( %EXPRESSION2 ) DO
        BEGIN
          %STATEMENT;
          %ID := %ID + 1
        END
      END .)
  ENDEF;

```

```

DEFINE %FACTOR
  RULE 'FACTORIAL' %FACTOR.INT
  MEANS
    EVAL:
      (. BEGIN
        LET @FACT := 1;
        *** SYNTAX ERROR: =          WANTED WHERE :=
        FOR @I := 2 TO %FACTOR DO @FACT := @FACT * @I
        -> @FACT
      END .)
  ENDEF;

```

PROGRAM

```

BEGIN
  LET @LIMIT = READI;
  FOR @I := 1 TO @LIMIT DO WRITEI FACTORIAL @I
END
EOF
***** ERRORS DETECTED *****

000.24 SECONDS IN EXECUTION
#EXECUTION TERMINATED
#

```

FOUN

Example 5

PROGRAM  
BEGIN

```
MACRO &FACTOR
  RULE ( 'READCH' ] 'READS' )
  MEANS EVAL: ( . READS( 1 ) . ) ENDMACRO;

MACRO &STATEMENT
  RULE 'WHILE' &EXPRESSION.BOOB 'DO' &STATEMENT.EVAL
  MEANS EVAL:
    ( . BEGIN
      LABEL @L;
      LAB @L:
      IF ( &EXPRESSION ) + 1 THEN
*** TYPE ERROR:
      BOOB FOUND WHERE EXPECTING INT
      BEGIN
        &STATEMENT;
        GOTO @L
      END
    END . )
ENDMACRO;

LET @CH = READCH;
WHILE @CH != " " DO
  BEGIN
    WRITES @CH;
    @CH := READCH
  END
```

END  
EOF

\*\*\*\*\* ERRORS DETECTED \*\*\*\*\*

000.20 SECONDS IN EXECUTION  
#EXECUTION TERMINATED  
#



Example 6

```

DEFINE &FACTOR
  RULE 'A' &NAME. (INT,PTR) 'NODE' &EXPRESSION1.INT
    'POINTING' 'TO' &EXPRESSION2.PTR
  MEANS
    EVAL: ( . CONSTRUCT &NAME( &EXPRESSION1, &EXPRESSION2 ) . )
  ENDEF;

DEFINE &STATEMENT
  RULE 'WHILE' &EXPRESSION.BOOL 'DO' &STATEMENT.EVAL
  MEANS
    EVAL:
      ( . BEGIN
        . LABEL @LOOP;
        LAB @LOOP: IF &EXPRESSION THEN
          BEGIN
            &STATEMENT;
            GOTO @LOOP
          END
        END . )
  ENDEF;

PROGRAM
BEGIN
  STRUCTURE @LIST( INT @INFO, PTR @NEXT ) ;
  STRUCTURE @NODE( INT @INF, PTR @LEFT, PTR @RIGHT ) ;
  LET @X = READI;
  LET @P = NULL;
  WHILE @X != 0 DO
  BEGIN
    @P := A @NODE NODE @X POINTING TO @P;
    *** TYPE ERROR:
      INT PTR PTR -CNSR FOUND WHERE EXPECTING INT PTR -CNSR
    @X := READI
  END;
  WHILE @P != NULL DO
  BEGIN
    WRITEI @INFO( @P ) ;
    @P := @NEXT( @P )
  END
END
EOF
***** ERRORS DETECTED *****

000.20 SECONDS IN EXECUTION
#EXECUTION TERMINATED
#

```

Example 7

```

REPLACE &STATEMENT
  RULE 'IF'
    BY 'IF' &EXPRESSION.BOOL 'THEN' &STATEMENT.EVAL
      @ELSEPART: (? 'ELSE' &STATEMENT.EVAL ?)
  MEANS
    GIVEN @ELSEPART THEN
      &STATEMENT:
        (. BEGIN
          LABEL @EXIT;
          IF &EXPRESSION THEN
            BEGIN
              &STATEMENT;
              GOTO @EXIT
            . END;
          &STATEMENT.@ELSEPART;
          LAB @EXIT
        END .)
      ELSE &STATEMENT: (. IF &EXPRESSION THEN &STATEMENT .)
    ENDEF;

REPLACE &STATEMENT
  RULE &VAR
    BY &VAR1.EVAL @VARS: (* ',' &VAR1.EVAL *) ':= '
      &EXPRESSION1.EVAL @EXPRS: (* ',' &EXPRESSION1.EVAL *)
  WHERE
    LIST
      ASSERT EOLEN( @VARS, @EXPRS ) : "UNBALANCED ASSIGNMENT." ,
      ASSERT MATCH( TYPE &VAR1, TYPE &EXPRESSION1 ) :
        "MISMATCHED TYPES FOR ASSIGNMENT." ,
      FORALL @VARS, @EXPRS:
        ASSERT MATCH(TYPE &VAR1.@VARS, TYPE &EXPRESSION1.@EXPRS) :
          "MISMATCHED TYPES FOR ASSIGNMENT."
    END
  MEANS
    LIST
      (. BEGIN &VAR1 := &EXPRESSION1 .) ,
      FORALL @VARS, @EXPRS:
        (. ; &VAR1.@VARS := &EXPRESSION1.@EXPRS .) ,
      (. END .)
    END
  ENDEF;

PROGRAM

BEGIN
  LET @X = READI;
  LET @Y = READI;
  LET @Z = READI;
  IF READI < @Z THEN @X, @Y, @Z := READI, @X, "NOT INTEGER" ELSE
  *** USER DEFINED ERROR: "MISMATCHED TYPES FOR ASSIGNMENT."
  **** ASSERTION UNSATISFIED.
  IF @Z=0 THEN @Z := READI
  ELSE WRITEI @Z
END

EDF
***** ERRORS DETECTED *****

000.30 SECONDS IN EXECUTION
#EXECUTION TERMINATED
#

```

Example 8

```

REPLACE &STATEMENT
  RULE 'IF'
    BY 'IF' &EXPRESSION4.BOOL 'THEN' &STATEMENT1.EVAL
      @ELSEPART: (? 'ELSE' &STATEMENT2.EVAL ?)
  MEANS
    GIVEN @ELSEPART THEN
      &STATEMENT:
        (. BEGIN
          LABEL @EXIT;
          IF &EXPRESSION4 THEN
            BEGIN
              &STATEMENT1;
              GOTO @EXIT
            END;
          &STATEMENT2.@ELSEPART;
          LAB @EXIT
        END .)
    ELSE
      &STATEMENT: (. IF &EXPRESSION4 THEN &STATEMENT1 .)
  ENDEF;

DEFINE &EXPRESSION
  RULE 'IF' &EXPRESSION1.BOOL 'THEN' &EXPRESSION2.EVAL
    'ELSE' &EXPRESSION3.EVAL
  WHERE ASSERT MATCH( TYPE &EXPRESSION2, TYPE &EXPRESSION3 ) :
    "MIXED TYPES FOR COND-EXPRESSION."
  MEANS
    (. BEGIN
      DECLARE @RESULT AS &EXPRESSION2;
      IF &EXPRESSION1 THEN @RESULT := &EXPRESSION2
      ELSE @RESULT := &EXPRESSION3
      -> @RESULT
    END .)
  ENDEF;

DEFINE &STATEMENT
  RULE 'CASE' &EXPRESSION.EVAL 'OF' 'BEGIN'
    @STMTS: (* &CONSTANT.EVAL ':' ' ' * ) &STATEMENT.EVAL ';'
    @OTHERS: (* &CONSTANT.EVAL ':' ' ' *) &STATEMENT.EVAL ';'
  'DEFAULT' ':' &STATEMENT.EVAL 'END'
  WHERE
    FORALL @STMTS:
      LIST
        MATCH( TYPE &EXPRESSION, TYPE &CONSTANT.@STMTS ) ,
        FORALL @OTHERS.@STMTS:
          MATCH( TYPE &EXPRESSION, TYPE &CONSTANT.@OTHERS.@STMTS )
    END
  MEANS
    LIST
      (. BEGIN LET @TEST = &EXPRESSION; .) ,
      FORALL @STMTS:
        LIST
          (. IF ( @TEST = &CONSTANT.@STMTS ) .) ,
          FORALL @OTHERS.@STMTS:
            (. OR ( @TEST = &CONSTANT.@OTHERS.@STMTS ) .) ,
            (. THEN &STATEMENT.@STMTS ELSE .)
          END,
          (. &STATEMENT END .)
    END
  ENDEF;

```

continued ...

... continuation of example 8

DELETE &STATEMENT RULE 'IF' ENDEF;

DELETE &STATEMENT RULE 'GOTO' ENDEF;

PROGRAM

BEGIN

LET @I = 0;

LET @J = 5;

LET @K = 10;

LET @X = IF READS( 3 ) = "NEG" THEN  
IF READI = READI THEN -1 ELSE 0  
ELSE IF READI = READI THEN 1 ELSE 0;

CASE READS(4) OF

BEGIN

"TRY1": "TRY2": WRITES "TRY1 OR TRY2";

"TRY3": @I := IF @X = 0 THEN READI ELSE @X;

DEFAULT : WRITES "NO TRYs"

END;

IF @X != 0 THEN WRITEI @X

\*\*\* SYNTAX ERROR: IF

NOT A STARTER FOR &STATEMENT

END

EOF

\*\*\*\*\* ERRORS DETECTED \*\*\*\*\*

000.45 SECONDS IN EXECUTION

#EXECUTION TERMINATED

#

Example 9

```

DEFINE &FACTOR
  RULE 'A' &NAME. (INT, PTR) 'NODE' &EXPRESSION1. INT
    'POINTING' 'TO' &EXPRESSION2. PTR
  MEANS
    EVAL: ( . CONSTRUCT &NAME( &EXPRESSION1, &EXPRESSION2 ) . )
  ENDEF;

```

```

DEFINE &STATEMENT
  RULE 'WHILE' &EXPRESSION. BOOL 'DO' &STATEMENT. EVAL
  MEANS
    EVAL:
      ( . BEGIN
        LABEL @LOOP;
        LAB @LOOP: IF &EXPRESSION THEN
          BEGIN
            &STATEMENT;
            GOTO @LOOP
          END
        END . )
  ENDEF;

```

```

PROGRAM
#CODE
BEGIN

```

```

  STRUCTURE @LIST( INT @INFO, PTR @NEXT ) ;
  STRUCTURE @NODE( INT @INF, PTR @LEFT, PTR @RIGHT ) ;
  LET @X = READI;
  LET @P = NULL;
  WHILE @X != 0 DO
    BEGIN
      @P := A @LIST NODE @X POINTING TO @P;
      @X := READI
    END;

```

```

      .. LAB 1
      .. LOADA 1
      .. CONTENTS
      .. LOADC 0
      .. NE
      .. JUMPF 2
      .. LOADA 2
      .. LOADA 1
      .. CONTENTS
      .. LOADA 2
      .. CONTENTS
      .. CONSTRUCT 0 2
      .. STORE
      .. LOADA 1
      .. READI
      .. STORE
      .. GOTO 1
      .. LAB 2

```

```

  WHILE @P != NULL DO
    BEGIN
      WRITEI @INFO( @P ) ;
      @P := @NEXT( @P )
    END

```

END

continued ...

... continuation of example 9

.. LAB	3
.. LOADA	2
.. CONTENTS	
.. LOADC	0
.. NE	
.. JUMPF	4
.. LOADA	2
.. OFFSET	1
.. CONTENTS	
.. WRITEI	
.. LOADA	2
.. LOADA	2
.. OFFSET	2
.. CONTENTS	
.. STORE	
.. GOTO	3
.. LAB	4
.. RESETSP	0
.. HALT	

EOF

000.23 SECONDS IN EXECUTION  
#EXECUTION TERMINATED  
#